AD NUMBER

ADB009033

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Gov't. agencies only; Test and Evaluation; 03 FEB 1975. Other requests shall be referred to Air Force Avionics Laboratory, Wright-Patterson AFB, OH 45433.

AUTHORITY

AFWAL ltr, 19 Oct 1981

AFAL-TR-75-242

A USER'S APPRAISAL OF AN AUTOMATED PROGRAM VERIFICATION AID

SYSTEM TECHNOLOGY BRANCH
SYSTEM AVIONICS DIVISION

DECEMBER 1975

TECHNICAL REPORT AFAL-TR-75-242

AIR FORCE AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
Air Force Systems Command
Wright-Patterson Air Force Base, Ohio 45433

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Distribution limited to U.S. Government agencies only (test and evaluation, 3 February 1975); data contains information which may be prejudicial to the manufacturer. Other requests for this document must be referred to AFAL/AAT, WPAFB, Ohio 45433.

This technical report has been reviewed and is approved for publication.

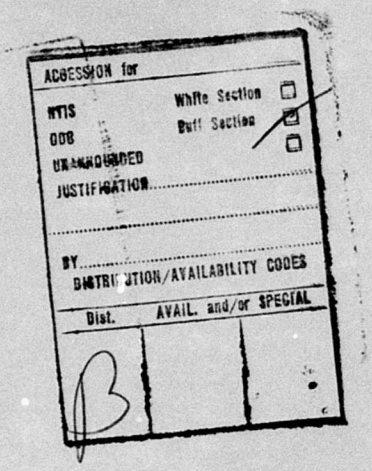
Larry K. Whipple

LARRY K. WHIPPLE, Captain, USAF
Project Engineer

FOR THE COMMANDER

George F. Cudahy

GEORGE F. CUDAHY, Colonel, USAF
Chief, System Avionics Division
AF Avionics Laboratory



Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

AIR FORCE - 30-1-76 - 100

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFAL-TR-75-242
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: A USER'S APPRAISAL OF AN AUTOMATED PROGRAM VERIFICATION AID.
5. TYPE OF REPORT & PERIOD COVERED: Final Report. Dec 74 - Oct 75
6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): LARRY K. WHIPPLE, MARK A. PITTS
8. CONTRACT OR GRANT NUMBER(s): 75 P.
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Air Force Avionics Laboratory (AFAL/AAT), Wright-Patterson AFB, Ohio 45433
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: AF - 2003-05-10; 2003P5
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Avionics Laboratory (AFAL/AA), Wright-Patterson AFB, Ohio 45433
12. REPORT DATE: Dec 75
13. NUMBER OF PAGES: 77
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*
15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution limited to U.S. Government agencies only (test and evaluation 3 February 1975); data contains information which may be prejudicial to the manufacturer. Other requests for this document must be referred to AFAL/AAT, WPAFB, OH 45433.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Approved for public release; distribution unlimited.

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Automated Verification Aids
Computer Program Test Tools
Computer Program Testing
Computer Program Verification and Validation
Computer Program Structure Modelling

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

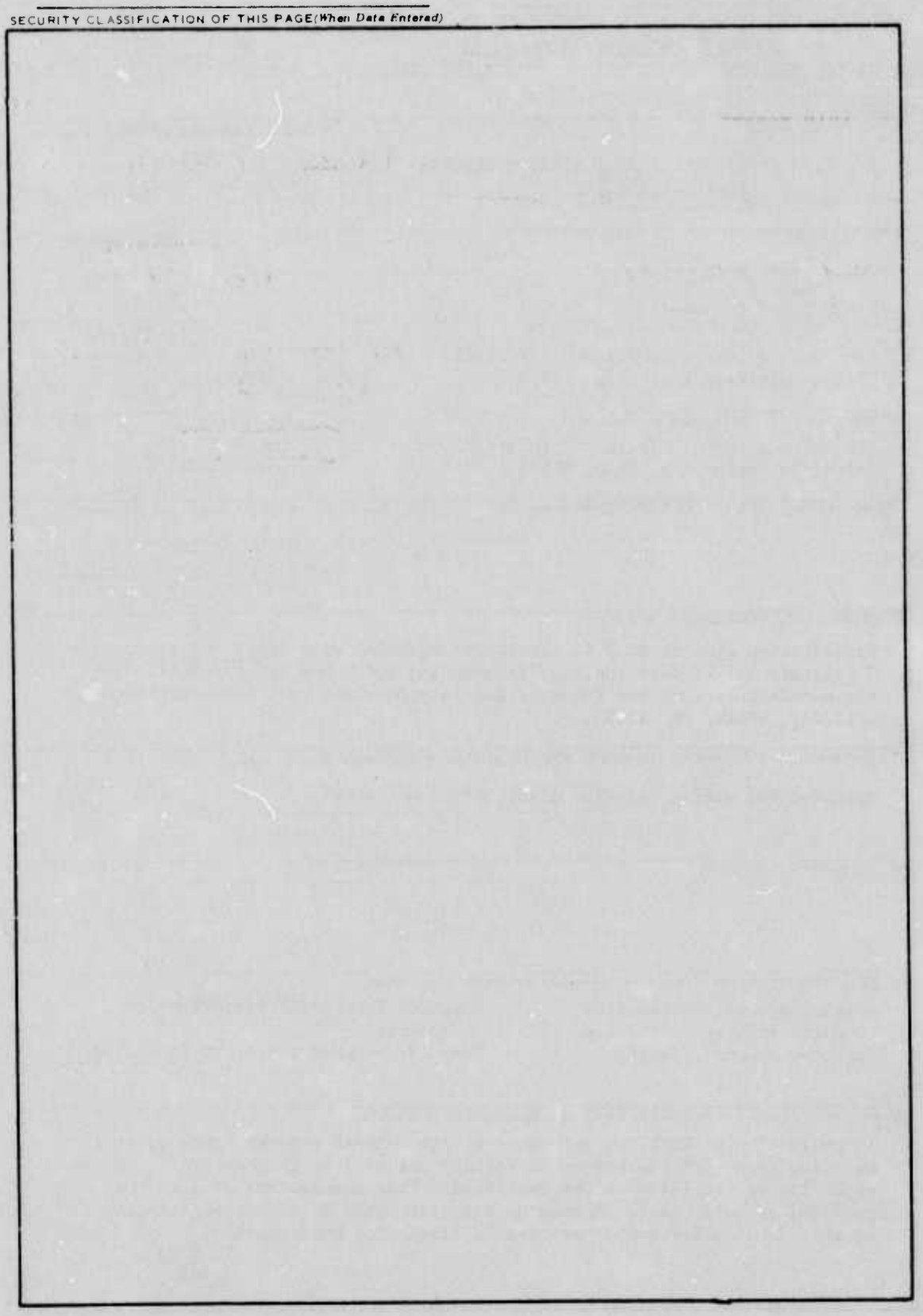
An evaluation of RXVP, an automated aid to FORTRAN program testing, based on experience with the system in testing activities, is presented. System capabilities and features are described. User assessments of services provided are discussed. Strengths and weaknesses in system performance as well as a collection of processing statistics are reported.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

011 670



PREFACE

This report reflects the results of work accomplished by the System Technology Branch, System Avionics Division, Air Force Avionics Laboratory, under Work Unit 2003-05-10. The project was sponsored by the Directorate of Information Systems, Headquarters, Air Force Systems Command (AFSC/ACD). Contractor support in connection with the project was provided by General Research Corporation, Santa Barbara, California, under Contract F33615-75-C-1195. The Air Force Project Engineer and Principal Investigator was Captain Larry K. Whipple (AFAL/AAT). Mr. Mark A. Pitts (AFAL/AAT) was Associate Investigator. The participation of the following individuals and organizations is gratefully acknowledged:

Dr. E. D. Callendar, Aerospace Corporation
Captain A. B. Carter, Sacramento Air Logistics Center
Mr. L. M. Culpepper, Naval Ship Research and Development Center
Mr. R. Daniel, U.S. Army Computer Systems Command
Lt. R. L. Ettenger, Space and Missile Systems Organization
Mr. C. W. Fowlks, Sacramento Air Logistics Center
Ms. M. A. Goodwin, NASA Johnson Space Center
Mr. R. A. Hansen, Sacramento Air Logistics Center
Lt. D. Herrington, HQ, Air Force Systems Command (ACD)
Mr. J. Palaimo, Rome Air Development Center
Ms. M. S. Plemmons, Defense Mapping Agency Aerospace Center
Ms. O. L. Power, Defense Mapping Agency Aerospace Center
Mr. R. Robinson, Rome Air Development Center
Captain J. L. R. Rooks, USAF Data Systems Design Center
Major A. W. Small, HQ, USAF (XOA)
Dr. R. B. Stillman, National Bureau of Standards

Thanks are also due Mrs. Sybil Hooper for her meticulous typing of the manuscript.

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 BACKGROUND

The military services, like our society as a whole, are becoming increasingly dependent on the digital computer to provide the information processing power necessary to function in an increasingly complex environment. When reliance is placed on a computer as a military decision aid or weapon system component, the reliability of that computer and of its software are matters of paramount importance. The cost of a critical failure or outage in this case may be measured not in terms of dollars or individual safety but in terms of national survival. For this reason, establishing the reliability of software developed or delivered for use in its systems is of prime importance to the Air Force.

Present technology does not permit "proving" the "correctness" of large software systems. Instead, a degree of confidence must be established through some evaluation/testing procedure. In testing its applications software, the Air Force at different times may assume either or both the roles of developer and acceptance tester [1]. As the developer of its own software, testing can sometimes proceed in a deductive manner based on knowledge of the internal logic of the program. As acceptance tester for software developed under contract, to the extent that the internal logic is unknown testing must proceed empirically. In neither case can absolute confidence be established, since complete testing of the program under all conditions is practically impossible. As stated by Gruenberger [2]: "... the art lies in knowing what to test for, how to devise adequate tests, and when to stop testing." The problem is one of designing the checkout and testing process to obtain maximum confidence in the program within the resources available.

The magnitude of resources involved is not small. An earlier, much-quoted report [3] estimates that roughly 45% to 50% of the total effort involved in a software project is typically spent in checkout and test activities. Further, that with current Air Force

expenditures of $1 billion per year for software, techniques saving one man-day of checkout and test activity per man-month would save $20 - $25 million per year.

As a result of all the factors mentioned, considerable attention has been focused on automated aids as ways to improve the efficiency of software development and of the testing activity in particular ([4], [5], [6]). In discussing automated aids, however, Reifer [4] concluded that:

> "For all practical purposes, there exist no answers to the following questions: What automated aids should we use and when in the life cycle should they be used? What are the effects of use? What are the limitations?"

This conclusion reflects the need for much more experience with such tools before conclusions may be drawn regarding their specific utility, the value of particular features, and optimum methods of application.

1.2 PROJECT DEFINITION

In late 1974, the Directorate of Information Systems, Air Force Systems Command (AFSC/ACD), decided to sponsor a project to examine an automated program testing aid from the user's point of view. A number of such aids were being advertised, interest in their use and benefits was high, but little information about them based on user experience was available.

The Air Force Avionics Laboratory (AFAL) was similarly interested in information about available automated program verification aids. Air Force avionics software verification and maintenance facilities, both proposed and established, require the support of such aids in accomplishing their functions. The utility and features of available tools were thus of interest from the standpoint of near term applications. In addition, areas found to be of particularly high potential benefit or areas found to be currently deficient could be used in planning research in avionics software verification technology. Finally, the Lab had in progress a number of software development projects in which it was felt an automated testing aid might be of

benefit. For these reasons, the Lab undertook the investigation sponsored by AFSC/ACD.

The project had two principal objectives:

(1) To examine the capabilities, utility, and features of a currently available automated testing aid from a user's point of view, and

(2) To provide testing support for AFAL and other software development projects.

A number of automated aids to program testing are available. Some have been developed as commercially marketable items in response to a perceived need. Others were developed (or are under development) to specification under contract. Still others grew out of program verification research projects. Several of the existing aids originated as a result of contractor efforts to automate the more tedious and error-prone manual processes associated with the software development portion of some contract. These aids were adapted and refined until enough general utility was obtained to permit the aid to be considered a generally applicable tool. (This development history is probably largely responsible for Reifer's [4] remark describing most automated aids as poorly structured, poorly documented, and poorly tested individual entities, not well integrated with each other and the people who use them.)

The verification aid selected for the project was RXVP (Level 1), developed by General Research Corporation (GRC), Santa Barbara, California. Several factors influenced this selection:

(1) RXVP was operational and available for immediate installation.

(2) RXVP was developed on the same family of computers as was to be used as host for the project. Transferability problems would thus be minimized.

(3) RXVP is advertised as being developed as "a complete, wholly integrated software verification system" offering "an organized approach to comprehensive testing" and supporting "the validation and verification effort through all its phases". RXVP could thus be considered as a total system in support of testing activities rather than as a collection of individual tools each providing support to some aspect of the testing process.

(4) RXVP was found to offer the broadest range of features of any verification system for which information was available.

(5) RXVP is based on a model of the iteration structure of FORTRAN programs. This same model forms the basis for a verification system for JOVIAL language programs (Jovial Automated Verification System) being developed for Rome Air Development Center. Examining RXVP and the use of this model might thus provide some advanced indication of the characteristics and capabilities of an automated testing aid soon to be a part of the Air Force inventory.

In brief, the project was organized around four principal activities. The contractor (GRC) installed and maintained the RXVP system on the CDC CYBER 74 computer at ASD Computer Science Center, Wright-Patterson AFB, Ohio. GRC also conducted a series of RXVP familiarization/training workshops for project participants and provided assistance to RXVP users as required. AFAL used the system in testing activities for avionics support software being developed to execute on both the CDC machine and another computer. A detailed examination of RXVP itself was then undertaken. To provide a greater variety of FORTRAN programs used in the project, a broader range of opinions on which to base conclusions, and the advice of those more experienced with testing tools, individuals from several government agencies having interest and/or experience in automated test tool technology were invited to participate in the workshops and to use the RXVP system on their own programs. These individuals made valuable contributions to the project. Finally, to demonstrate expert application of the RXVP system in testing a program not familiar to the tester (as in the role of acceptance tester), GRC was to test a selected program developed by AFAL.

1.3 REPORT ORGANIZATION

The balance of this report discusses the details of the work accomplished during the project and the results obtained.

Section 2.0 provides an overview of the RXVP system to familiarize the reader with pertinent terminology and characteristics.

Section 3.0 presents the project objectives and procedures in more detail.

Section 4.0 contains the results and conclusions derived from the project activities.

Section 5.0 summarizes salient conclusions about RXVP and automated testing tools in general, and presents recommendations based on those conclusions.

No familiarity with the RXVP system is required for reading this report at a general level. To understand some of the more specific comments and references, however, requires a degree of familiarity obtainable by reading the RXVP User's Guide and Reference Manual ([9], [10]).

2.0 RXVP OVERVIEW

2.1 RXVP OBJECTIVES

RXVP is an automated software verification aid designed to assist in the test and verification of FORTRAN programs. It performs a structural analysis of the subject program and stores the results in a data base. Using the data base, RXVP can supply:

- static analyses of individual program modules and groups of modules
- automatic instrumentation of the program control structure
- instrumentation at the statement level for recording statistics on program variables
- testing strategy guidance
- assistance in generating test cases
- quick-look and post-test analyses of testing coverage
- post-test reports of program variable statistics.

The objective of RXVP is to provide analysis services needed for the verification and validation of large FORTRAN software systems of up to one-thousand modules totaling up to two-hundred-fifty-thousand statements.

RXVP is intended to support the verification and validation effort by promoting the systematic testing of single modules or groups of modules. It provides the user with (1) test coverage documentation, detailing exactly what portions of the program modules were exercised, (2) test case generation assistance to help the user generate test cases to exercise untested portions of each module, and (3) static analyses providing information helpful in locating possible sources of program error.

2.2 RXVP ORGANIZATION

RXVP is organized into ten separate STEPs, each of which conducts a functionally related set of processing tasks, which communicate through a common data base. These STEPs and their associated functions are listed below:

STEP 1 (BASIC)	Source text input, lexical scan, syntactic recognition, and initial source library creation
STEP 2 (STRUCTURAL)	Structural analysis and execution path identification; library update with structure and path information
STEP 3 (INSTRUMENT)	Program instrumentation for execution path coverage analysis
STEP 4 (QUICKLOOK)	Execution of instrumented code and quick-look analysis of program path coverage
STEP 5 (ANALYZER)	Detailed analysis of program path coverage; execution traces and summary statistics
STEP 6 (SELFMET)	Statement-level instrumentation for program performance analysis
STEP 7 (SMANALYZE)	Detailed analysis of program performance with individual statement execution results
STEP 8 (STATIC)	Program static analysis; subroutine call sequences; array subscript checks; expression mode checks; etc.
STEP 9 (GUIDE)	Test guidance providing rational, systematic testing strategy not immediately visible from inspection of source text
STEP 10 (ASSIST)	Assistance in the generation of test cases to exercise untested program paths.

It should be noted here that all these STEPs need not, and probably will not, be executed in the above order. Neither will they all necessarily be executed in any one testing activity.

Included in many of the processing STEPs is a data manager that provides library merging capabilities, report heading and subheading specifications, initialization commands processing, module selection, standard print-outs of library contents, and specification of alternative files to be used during RXVP operation.

RXVP uses a command language to provide control of the standard processing STEPs. Certain commands are known as "universal" because they are common to all STEPs while other commands are only recognized by certain STEPs.

A random access file is used to record the original source text and associated tables containing structural information, symbols and their classification, and other module-descriptive information. This random access file, known as the library, is created during the first processing step (STEP 1). During the next step (STEP 2), the library is updated to include program structure information. After STEP 2 processing, the library is complete and is used as input for all remaining processing STEPs. STEPS 1 and 2, therefore, need not be run again as long as the library is saved. The major sections of the completed library are:

(1) Analyzed source text
(2) Module descriptor block
(3) Entry points
(4) Statement descriptor block
(5) Symbol table
(6) Program structure information

2.3 RXVP OPERATION

2.3.1 How RXVP is Used

The program modules to be analyzed by RXVP are assumed to compile correctly using a language translator for the FORTRAN dialect specified as the one RXVP is installed to accept, and to execute to termination for some given set of initial test data.

RXVP processing is organized into four major phases. The first phase is the basic processing of the source code. This phase reads the modules, builds the symbol and statement tables, assigns node and statement numbers, identifies module structure in terms of DD-paths and level-i paths (see Paragraph 2.4), and provides a comprehensive static analysis. All this information is then assembled into the data base or library.

The second phase accomplishes the instrumentation of the control structure and/or individual statements of each program module.

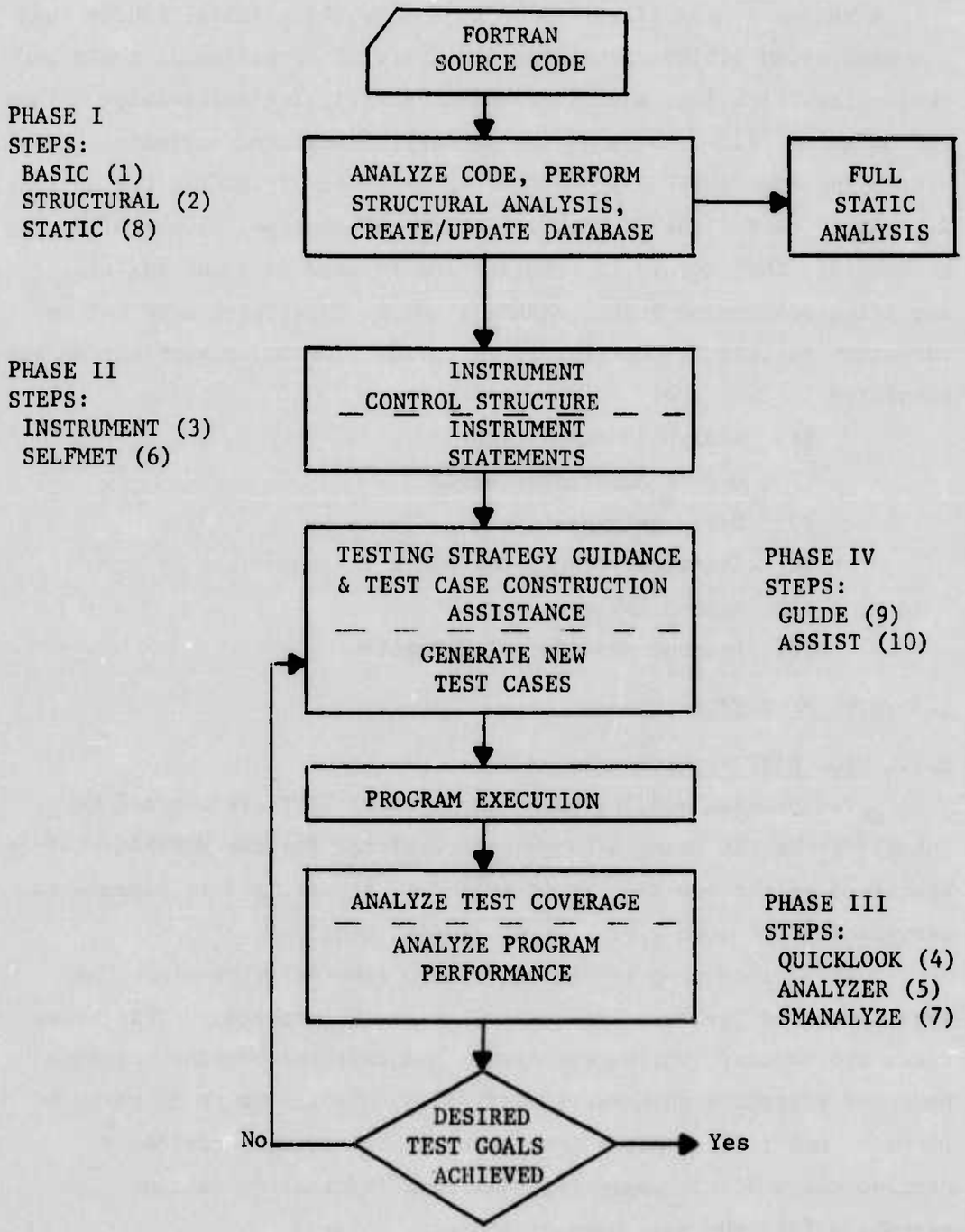


Figure 2.1 RXVP Testing Overview

The third phase is the test coverage analysis phase. It provides detailed analyses of control structure and individual statement coverage, and reports on the computational behavior of program variables.

Phase four generates reports on overall testing strategy and reports to aid in the generation of new test cases.

Figure 2.1 gives an overview of how RXVP is used in the testing of software and lists the STEPs for each phase.

Each STEP contains execution options, selectable using the RXVP command language, to tailor the processing to the user's needs.

As previously stated, not all of the STEPs (and certainly not all of the options) need to be executed in the process of testing a software system. STEPs 1 and 2 must be run to create and update the program library. It is then left to the user to determine the STEPs and options that will provide the information and services relevant to his individual testing needs.

Each RXVP run begins with a set of STARTUP commands which directs RXVP initialization and ends with the command START. Next come the commands describing the STEPs and options which are to be executed. These commands are followed by the END command, which correctly terminates the RXVP run by closing the appropriate files. The reader is referred to References [9] and [10] for more details regarding use of the RXVP system.

2.3.2 Modes of Operation

Operating modes of individual installations of RXVP are largely dependent on the facilities and operating procedures of the host computer system. Factors affecting operating modes of an RXVP installation include computer memory size, mass storage capacity, operating system features, job scheduling algorithm, and facility administrative practices.

In some installations, each RXVP processing step constitutes a separate job or job step. In other installations, RXVP is installed

as a system of overlays, with all (or selected) processing steps accessible by command in a single job step.

2.4 FORTRAN PROGRAM ITERATION STRUCTURE MODELING

The structural analysis performed by RXVP for each program module consists of (1) determining all Decision-to-Decision paths (DD-paths) within the module and (2) combining DD-paths into non-iterative sequences called level-i paths.

A DD-path is a sequence of statements between decision points in a module. It begins with the sensing of the result of some predicate evaluation and includes all subsequent statements through the evaluation of the next predicate, but not the action taken as a result of the evaluation of the second predicate.

```
                    .
                    .
                    .
IF(I.EQ.10) GO TO 100          (10,11)
                    .
                    .
                    .
IF(N.GT.15) K = 20             (12,13)
J = I + 20
                    .
                    .
                    .
100 CONTINUE
                    .
                    .
                    .
```

Figure 2.2 DD-Path Illustration

In Figure 2.2, DD-paths 10 and 11 begin with the result of the evaluation of the predicate (I.EQ.10). DD-path 10 represents the TRUE result, commencing with the statement GO TO 100, and including the CONTINUE and any subsequent statements through the evaluation of the next predicate.

DD-path 11 represents the FALSE result, commencing with the statement following the IF-GO TO statement and ending with the evaluation of the predicate (N.GT.15).

DD-path 12 begins with a TRUE result of the evaluation of the predicate (N.GT.15) and includes the statement K = 20 as well as subsequent statements through the evaluation of the next predicate. DD-path 13 commences with a FALSE evaluation of (N.GT.15), continues with the statement J = I + 20 (omitting K = 20), and terminates with the evaluation of the next predicate. Note that DD-paths 12 and 13 both begin and end at the same points in the module and thus can be considered to be parallel.

A DD-path class is a set of parallel DD-paths (called BROTHERS). In Figure 2.2, DD-paths 12 and 13 are parallel (hence, are BROTHERS) and are members of the same DD-path class, while DD-paths 10 and 11 are non-parallel and are not members of the same DD-path class.

Level-i paths are sequences of DD-paths which comprise non-iterative flows in the program module. A level-0 path is a sequence of DD-paths that begins at module entry, ends at module exit, and does not traverse any decision point more than once. (See Figure 2.3.)

A level-1 path is a sequence of DD-paths that begins at a decision point on a level-0 path, ends at the same or an earlier decision point on the same level-0 path, and does not include any DD-paths which are on level-0. A level-2 path is similarly defined relative to level-1 and so on for higher order level-i paths.

This constructive definition permits iterative flows through the program to generally be described in terms of combinations (NB not sequences) of level-i paths.

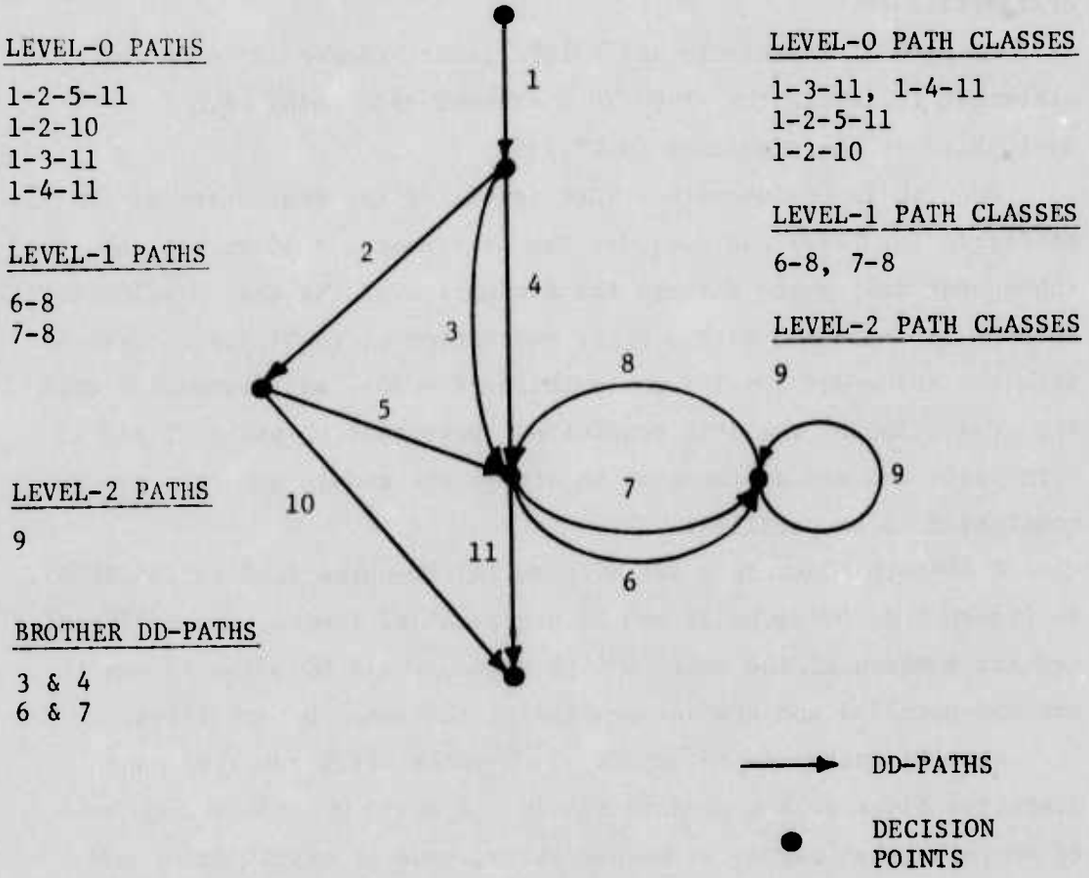


Figure 2.3 Directed Graph of a Sample Program Structure

A level-i path class, i=0,1,2,..., is a set of level-i paths, each a different sequence of DD-paths from the same collection of DD-path classes, that traverse the same set of module decision points.

The structure of a module can be represented by a directed graph in which nodes represent decision points in the module and edges represent DD-paths. (Figure 2.3) See references [7], [8], [9], and [10] for a more complete discussion of iteration structure modelling.

3.0 PROJECT OBJECTIVES AND PROCEDURES

3.1 OBJECTIVES

As previously mentioned, the project was designed to accomplish two principal objectives:

(1) Examine the capabilities, features, and utility of a currently available automated program verification aid (RXVP) from the user's point of view, and

(2) Provide testing assistance to AFAL and other software development projects.

The examination to be conducted in order to satisfy the first objective consisted of two parts. The first part was a subjective evaluation of RXVP based on the experience gained in providing testing assistance to the software development projects (i.e., in accomplishing the second objective). This evaluation was to assess the benefits derived from the use of RXVP in the testing function, the convenience with which RXVP could be applied in the user's testing environment, and the adequacy of the features provided for accomplishing the testing objectives. Also to be considered were the amount of user familiarization and training required to apply RXVP beneficially in the testing process, the adequacy of the user documentation supplied with the tool, and the ease of interpreting the testing and analysis results.

The second part of the examination was an objective investigation of certain characteristics of the RXVP system. The goal here was to obtain such measurements as the execution time required for various RXVP analysis functions, the amount of core memory required for RXVP execution, the amount of on-line storage required for RXVP installation and RXVP data bases, and the core and execution time expansion factors resulting from RXVP instrumentation of a module under test. Discrepancies in RXVP performance were also to be identified and investigated in this part of the examination.

Four of the five criteria set forth by Ramamoorthy and Ho [6] for evaluating automated software support tools were addressed by this two part examination. The subjective evaluation principally

addressed the criteria of resolution power (i.e., the facility with which the tool provides interesting details extracted from the bulk of the source code) and ease of use. The objective part was mainly concerned with the criteria of tool validation (i.e., how well the tool itself has been qualified) and, to some extent, with transferability and flexibility characteristics for adapting to different testing environments. The criterion of generality, reflected in an ability to accept different languages, was not addressed, although the ability to accept different dialects of the same language (FORTRAN) was considered.

3.2 PROCEDURES

3.2.1 RXVP Installation

RXVP was installed at the ASD Computer Science Center on a CDC CYBER 74 computer to operate under the SCOPE 3.4 operating system. The system was installed as a series of overlays so that all RXVP analysis STEPs were accessible (via the command file) from a single invocation of the RXVP program. The usual operating mode was remote batch from a terminal located at AFAL. Installation specifications called for RXVP to process FORTRAN programs written in the CDC FORTRAN Extended dialect as well as the ANSI X3.9 Standard FORTRAN.

A modified version of RXVP STEP 4 (QUICKLOOK) was installed on a Datacraft 6024/4 computer at AFAL. The Datacraft 6024/4 is a 24-bit/word machine with a 750 nanosecond cycle time. The memory size at the time of the RXVP installation was 32K words. Mass storage consisted of a single nine-track tape drive and a Datacraft 5208 Cartridge Disk Drive with a capacity of approximately 2.75 million words. The Datacraft is being used as a system component in an avionics software support system being developed by AFAL. The purpose of the installation on the Datacraft was to examine the use of RXVP in testing software executing on a machine other than the RXVP host machine. This situation would be common when using a verification aid in conjunction with avionics software development and maintenance. Considering the characteristics of computers often used as system

components, it is highly probable that verification aids used in testing software developed for most systems would not be installed on the computer where the testing would be accomplished.

The principal difference between the version of STEP 4 installed on the Datacraft and the version installed as part of the RXVP system on the CYBER 74 was the manner in which the QUICKLOOK testing coverage reports were generated. In the standard version, testing coverage reports are generated automatically following each invocation of the instrumented program. Provisions are made for repeatedly invoking the instrumented program to permit several test data-sets to be processed as a group [10]. This requires the report generation and invocation control routines, as well as the execution trace data collection routine, to be loaded with the instrumented program under test. To minimize the amount of memory required to execute instrumented programs on the Datacraft, only the execution trace collection routine was loaded with an instrumented program. Execution coverage reports were produced in a subsequent run from the trace file recorded during execution of the instrumented program. A separate Testing History file containing summaries of coverage achieved by each execution was referenced and updated by each report generation run. This file permitted cumulative coverage reports for several test data-sets to be produced.

3.2.2 Non-Host Computer Program Testing

Two avionics software support programs, written in ANSI X3.9 FORTRAN to execute on the Datacraft 6024/4, were selected for testing: a generalized table-driven assembler (ALAP) and an instruction-level computer simulator (ILS). These programs consist of 1366 FORTRAN statements in 24 modules and 1604 FORTRAN statements in 32 modules respectively. The type of testing to be accomplished was of the final delivery/acceptance category. Testing goals were to exercise the known principal logic flows using standard test data sets, to measure the execution coverage provided by these data sets, then to achieve execution coverage of all DD-paths in the program. The assistance

of RXVP in creating program documentation was also of interest. The basic procedure established for testing each of these programs was as follows:

(1) The program was processed through RXVP STEP 1 (BASIC) and STEP 2 (STRUCTURAL) on the CYBER 74 and the RXVP data base created was cataloged for future reference.

(2) Any desired RXVP static analysis reports were produced.

(3) The program was instrumented for execution coverage data collection by RXVP STEP 3 and the instrumented source deck was punched.

(4) The instrumented source deck was compiled and executed on the Datacraft using the standard test data sets.

(5) Execution coverage reports were generated on the Datacraft from the trace file recorded during the previous execution and from the Testing History file, the Testing History file was updated, and the cumulative execution coverage was determined.

(6) Testing guidance and test case construction assistance reports to assist in accessing DD-paths not yet executed were obtained from the CYBER 74.

(7) The instrumented program was again executed on the Datacraft with test data derived using the reports produced in Step (6).

(8) Steps (5) through (7) were repeated as necessary.

ILS was to be tested by GRC to demonstrate expert application of RXVP in testing a program with which the tester (i.e., acceptance tester) was not intimately familiar. ALAP was tested by AFAL.

3.2.3 Program Testing on the RXVP Host Computer

A collection of programs of various types (e.g., simulation models, language processors, scientific/numerical calculations, etc.) was analyzed and tested on the CDC CYBER 74 using RXVP. The types of testing ranged from development and debug testing through acceptance testing. Some of these programs were developed by AFAL. Others belonged to organizations invited to participate in the familiarization workshops and/or other aspects of the project. Programs examined during the familiarization workshops were generally restricted to 400 statements or less due to the limited amount of time

available and the difficulties sometimes encountered in transferring programs to the CYBER 74. Each participant used RXVP however he desired in examining his programs. Most used procedures very similar to the one illustrated in Figure 2.1.

Comments were solicited from the project participants regarding their experiences and impressions. These comments together with the conclusions of the AFAL investigators formed the basis of the subjective evaluation of RXVP.

3.2.4 Investigation of RXVP Characteristics

To measure various characteristics of interest to potential users of RXVP, a subset of the programs previously examined was selected and subjected to a standardized RXVP processing procedure. An attempt was made to select a mix of programs of differing types, sizes, structures, and complexities. The objective was not to obtain a valid statistical characterization of RXVP performance. Rather, it was to obtain order-of-magnitude indications of the values of certain performance factors in RXVP analysis of assorted real programs.

Each selected program was first compiled and, where possible, executed to obtain core and execution time baselines. All measurements were taken on the CYBER-74. This was done primarily for two reasons: to reduce the number of additional variables involved when two computer systems are used (e.g., different compiler efficiencies, different loader characteristics, etc.), and because it was not convenient to measure execution time with sufficient precision on the Datacraft. The programs designed for the Datacraft could not be executed on the CYBER-74 due to word-length dependencies. No execution time measurements were therefore made for these programs.

After obtaining the baseline measurements, each program was processed through RXVP with the following options selected for each module in the program:

(a) Basic analysis

(b) Structural analysis

(c) Print module and symbol table

(d) Full static analysis

(e) Print DD-Paths (DDP's) and Level-i Path Classes (LIP*'s)

(f) Print the program graph

(g) Print summary of module characteristics and history of analysis results

(h) Instrument the module for testing coverage data collection

(i) Instrument the module for statement-level execution data collection.

Control statements in the RXVP command file caused RXVP execution times to be reported for each command (or closely related group of commands) as each module was processed.

Compilation of the instrumented program decks to determine new core requirements was the last step in processing each of the unexecutable programs.

For each of the executable programs, the deck (instrumented for testing coverage data collection) was compiled and executed under control of the RXVP STEP 4 (QUICKLOOK) testing option. Here, RXVP controlled automatic invocations of the instrumented program for each test data-set identified, recorded a trace file of the resulting executions, and generated individual and summary (cumulative) reports for each invocation of the program.

The next step was RXVP analysis of the execution trace file just recorded, and correlation of that data with data in the RXVP data base describing the structure and complexity of each module in the program.

The program deck (instrumented for statement-level execution data collection) was next compiled and executed to generate a statement-level trace file.

The final step was RXVP analysis of the statement-level trace file.

The combination and order of options selected for the RXVP processing of each of the programs was not intended to reflect typical use of the system in any real application. The purpose was to obtain the desired performance measurements for a variety of

programs. In some cases, therefore, some of the outputs were included for completeness and standardization rather than because they contributed important information concerning the particular program being processed.

As a result of the above standard processing, the following measurements were obtained using the selected sample programs:

(1) RXVP execution times (on a per-module basis) for basic analysis (STEP 1); full structural analysis (STEP 2) with all reports; printing of the analyzed module and its symbol table; full static analysis (STEP 8); printing of all DDP's, LIP*'s, and the module structure graph; printing the summary of module characteristics and history of analysis results; instrumenting the module for testing coverage data collection; and instrumenting the module for statement-level execution data collection.

(2) RXVP execution times (on a per program basis) for QUICKLOOK (STEP 4) report generation, STEP 5 analysis of execution coverage trace file, and STEP 7 analysis of statement-level trace file.

(3) Module core expansion factor and program execution time expansion factor resulting from both execution coverage and statement-level instrumentation.

(4) The number of words of on-line storage required for the RXVP data base.

The amount of core required to execute RXVP itself and the core overhead connected with execution of instrumented programs were also determined.

Finally, a number of questions which arose during the testing process were investigated using modules specifically designed for this purpose. (This generally involved isolating and identifying some RXVP performance anomaly.)

4.0 RESULTS AND CONCLUSIONS

The results obtained from the examinations conducted during the project and the conclusions made based on those results are presented in this section. The section is organized following the structure of RXVP. Each RXVP functional STEP is first discussed individually. The function of the STEP is briefly reviewed and any subjective user reactions to the STEP are summarized and discussed. These reactions generally reflect the user's evaluation of the STEP in terms of such factors as ease of interpreting output format and content, adequacy of the features incorporated for accomplishing the intended function, and problem areas encountered in using the STEP. Next follows a discussion of the STEP in more objective terms. Any deficiencies and/or discrepancies encountered in using the STEP are presented. Here a deficiency is defined to be any system fault or limitation, whether documented or not in RXVP publications, which was felt to adversely affect system utility. A discrepancy is defined to be any system characteristic in conflict with either RXVP documentation or contract requirements. (Several deficiencies/discrepancies were corrected by GRC during the course of the project. Since our objective here is, insofar as possible, to discuss the system as it currently exists, and since these corrections are presumably incorporated into any future installation, these items are not explicitly referenced.) Finally, any appropriate general statistics which were found to characterize STEP operations are presented. (Statistics collected during standard processing of specific modules are reported in the Appendix.)

Following individual consideration of the RXVP processing STEPs, items pertaining not to specific STEPs but to the system as a whole are discussed. The RXVP command language and system documentation are two of the topics included in this discussion. Finally, RXVP installation considerations are addressed. These considerations deal not only with physical facilities required, but with system

availability to the user and its effect on the manner in which the system is used.

4.1 RXVP STEP 1 (BASIC PROCESSING)

The function of STEP 1 is to input a FORTRAN program from the RXVP INPUT file and convert it to a series of descriptive tables to be stored in a random-access library file for future reference. These tables, the statement descriptor table, the statement table, and the symbol table, are the principal outputs from the STEP. User control options specify whether or not program COMMENTS are to be retained in the library file, whether the program is to be listed or not, and how many modules on the INPUT file are to be processed. Output to the user consists of the module number assigned, some storage and processing time statistics, a few basic module statistics, the module listing (if selected), and certain error messages if difficulty is encountered in processing the module.

BASIC processing is accomplished in two stages. The first stage is a lexical scan of the module wherein statement tokens are isolated and identified. In the second stage, the statements are parsed, classified, and node numbers are assigned.

Since the function of STEP 1 is not primarily user oriented, user subjective reaction to the STEP was minimal. The only comment concerned the way in which statement tokens are isolated in storing the module in the library file. Blanks inserted to isolate tokens sometimes cause subsequent listings of the module to be more difficult to read.

STEP 1, being the FORTRAN "recognizer" for the RXVP system, implicitly defines the set of programs which can be processed. Any deficiencies or discrepancies in STEP 1 are therefore significant because they limit this set.

4.1.1 STEP 1 Deficiencies Noted

(a) FORTRAN keywords, if not delimited by a special character, must be blank delimited. This constraint is documented in the RXVP Reference Manual. Its impact when analyzing a particular program is obviously a function of programming

style. Its most serious consequence in our experience resulted from failure to recognize DO statements of the form: DO20J = 1,N. Because all vital RXVP functions are based on a model of the iteration control structure of the module, failure to recognize such an important component of that structure is catastrophic.

(b) No statement may contain more than 250 tokens. This constraint is documented in the RXVP Reference Manual which suggests breaking the statement into equivalent smaller statements. This was philosophically objectionable to all those who encountered the problem, as the general opinion was that the tool should accommodate the program and not vice-versa. The most common way in which the constraint was violated was in long DATA statements, which required considerable care to segment correctly.

(c) Any EQUIVALENCE statement must follow all COMMON and DIMENSION statements. This constraint is also documented in the RXVP Reference Manual. It imposes an artificial restriction on programming style. (Whether any particular restriction was good or bad was not addressed. The criterion used was whether any standards were imposed in addition to those required to compile correctly.) Several programmers held the view that the EQUIVALENCE relation was clearer when stated in conjunction with the declaration/dimensioning of the variables involved.

(d) Only single expression IF statements are permitted. This constraint is documented in the RXVP Reference Manual. For the set of programs used during the project it caused few problems, since the IF-IF construct was encountered only rarely.

(e) COMMENTS consisting of more than 65 characters following the C (in Column 1 of the card) are truncated to 65 characters. COMMENTS are realigned by RXVP so that the first non-blank character following the C is placed in Column 7. If the COMMENT was purposely aligned differently by the programmer for some reason, this realignment defeats his purpose.

4.1.2 STEP 1 Discrepancies Noted

(a) The installation specification called for RXVP to process FORTRAN programs written in the CDC FORTRAN Extended (Version 4) dialect. (This dialect is hereafter referred to as FTN.) It was soon determined that the system as installed was not compatible with FTN, principally because of the instrumentation inserted by STEP 3 (see Section 4.3.2, Item (a)). STEP 1 was also found not to recognize certain FTN constructs (e.g., the alternate

RETURNS(S1,..,Sn), RETURN i, and READ fn,varlist constructs). The installation was, in fact, found to be more compatible with the CDC RUN compiler dialect. Since for the purpose of the project either dialect was generally satisfactory, a decision was made to adopt RUN as the standard dialect in order to proceed most expediently. No attempt was made to systematically determine all the STEP 1 incompatibilities with either the FTN or RUN dialects. Instead, whenever an incompatibility was detected which involved a language feature common to both RUN and FTN it was recorded. The decision to adopt the RUN dialect should not obscure the fact that RXVP as installed did not successfully accommodate the specified FORTRAN dialect.

(b) Interior blanks are not permitted in any symbol. The occurrence of interior blanks can cause spurious variable names to be generated (some even defining variables as having numeric names), misinterpretation of FORTRAN keywords and spuriously generated keywords, logical and relational operators to be unrecognized, and numeric values to be split into parts with resultant difficulty in recognizing the mode of the constant.

(c) Multiple statements per card are not permitted. The statement separator "$" is not recognized.

(d) The n.Dm and n.D-m data specifications are not recognized as double-precision. They are classified instead as type REAL.

(e) Double-precision constants are truncated to 10 characters. The truncated constant cannot then be recognized in many cases.

(f) An array dimensioned in an INTEGER statement and subsequently assigned to COMMON by a COMMON statement is identified as a LOCAL rather than COMMON array.

(g) COMMENTS between continuation cards are not permitted. This feature of FTN and RUN is sometimes used by programmers to label the data items in a long DATA statement.

(h) The NAMELIST statement is not recognized by the parser.

4.1.3 STEP 1 Statistics

RXVP STEP 1 required approximately 69,000 words of CYBER-74 main memory for execution. This includes provision of the Standard Print Commands for printing the library tables produced in STEP 1.

4.2 RXVP STEP 2 (STRUCTURAL ANALYSIS)

The function of STEP 2 is to perform the analysis of program module iteration structure on which the remaining RXVP STEPs are based and to add this structural information to the RXVP library file established in STEP 1. The analysis performed by STEP 2 consists of (1) identifying all DD-paths in the module, (2) determining the module iteration structure in terms of level-i paths, and (3) computing a series of complexity measures based on the types of statements comprising the module and its iteration structure.

The module complexity measures are computed using three factors: individual statement complexity, DD-path complexity, and level-i path complexity. The complexity of a single statement is found by adding a constant value (reflecting the statement type) and the length of any expressions in the statement. Expression length is defined to be the number of tokens required to write the expression using reverse Polish notation. The sum of all constituent statement complexities defines the "total static complexity" of a module. The complexity of a DD-path is computed from (1) the sum of the complexities of the statements comprising the DD-path and (2) a weighting factor $W_L = 2^{**}L$, where L is the (highest) iteration level on which the DD-path lies. The sum of all DD-path complexities is called, naturally enough, the "total DD-path complexity" of the module. Finally, the "total level-i path class complexity" is defined to be the sum of the complexities of all level-i path classes in the module. The complexity of each level-i path class is found by adding the DD-path complexities for a representative member of the level-i path class.

Control options permit the user to specify whether DD-paths alone or DD-paths and level-i paths are to be identified, whether complexity measures are to be computed, and whether a summary report or one detailing the individual statement, DD-path, and level-i path class complexities is to be produced.

According to RXVP documentation [10], the complexity estimates made by STEP 2 provide a basis for rationalizing decisions regarding

where to next apply testing effort in a testing activity. The hypothesis is that the more complex portions of a software system (as identified by the various complexity measures) should receive testing priority.

In our experience, RXVP users paid little attention to the complexity measures. One reason was an inability to correlate the complexity measures computed by RXVP with one's own judgmental estimate of module complexity. This, of course, is because the number of factors taken into account by an individual in estimating the ill-defined property of module complexity is much larger than that used in the RXVP algorithms. The (subjective) weights assigned to the various factors are very much a function of the individual's past experience, and may be influenced by his knowledge of how the particular module is to be used. A few sample remarks are indicative of the general lack of regard for the RXVP complexity measures:

(1) Subroutine CALLs and FUNCTION invocations are known to contribute to the probability of errors in execution due to interface problems and side-effects. Because they represent relinquishing control of execution (and hence control of future computational environment) by a module, they serve to increase the module's "complexity", if in that term one includes the interaction of a module with its environment (i.e., modules with higher degrees of "connectivity" are more "complex" than those without). In computing statement complexity, however, RXVP algorithms give no more weight to CALLs or FUNCTION invocations than to simple assignment statements.

(2) The complexity of a level-i path class is computed using the "leader" (i.e., lowest-numbered) DD-path from each set of parallel DD-paths in the level-i path class. Since the "leader" DD-path is not necessarily the most complex member of a parallel set, it may be a poor basis for determining the complexity of the level-i path class. More important, by using a single level-i path (the one consisting of "leader" DD-paths) to determine the complexity of an entire level-i path class, no distinction is made between a level-i path class with a single member (no parallel DD-paths) and a level-i path class with multiple members (representing alternate flows of execution within the class).

(3) What is the justification for weighting the static (total statement) complexity of a DD-path with an exponential function of the iteration level on which the DD-path lies in determining DD-path complexity? That is, why does nesting a given DD-path one level deeper in an iteration structure double its complexity?

Most RXVP users who considered the complexity measures at all regarded them as interesting numbers, uniformly computed for various modules, but of little practical significance in program testing.

While complexity measures are computed in STEP 2, the primary function of the STEP is to develop the structural model of the selected FORTRAN module. This model, expressed in terms of DD-paths and level-i paths as discussed in Section 2.4, is the basis for the remaining RXVP STEPs concerned with module control structure instrumentation and testing. Most users, accustomed to thinking of modules in terms of sequential flows of execution, had little difficulty understanding the concept of DD-paths. Level-i paths and level-i path classes, however, representing a less familiar decomposition of the module into levels of iteration, generally proved more difficult to understand.

4.2.1 STEP 2 Deficiencies Noted

(a) The only deficiency noted in the ability of STEP 2 to model the iteration structure of FORTRAN modules is attributed not to a problem in STEP 2 per se but in the relation between the definition of level-i path and potential iteration in the module. According to the RXVP Reference Manual [10], the level-i path concept is intended to permit defining flows within a program in terms of successive "levels" of iteration. Quoting from that source:

> "Level-i paths are non-iterative sequences of DD-paths defined in such a way that flows through the program (including iteration) can be represented as combinations of level-i paths . . . a level-0 path is a sequence of DD-paths that begins at program entry and ends at program exit, and is non-iterative; that is, it does not traverse any decision point more than once . . . A level-1 path is a non-iterative sequence of DD-paths, none of them on a level-0 path, that begins at a decision point of a level-0

path and ends at the same or an earlier decision point of the same level-0 path."

A level-i+1 path thus represents a program flow which forces a repetition of some decision that lies on a level-i path. Since level-i paths are defined strictly on the basis of structural considerations, prohibiting any DD-path which is part of a level-i path from being a part of a level-i+1 path (by definition) renders the model incapable of representing certain types of iteration structures.

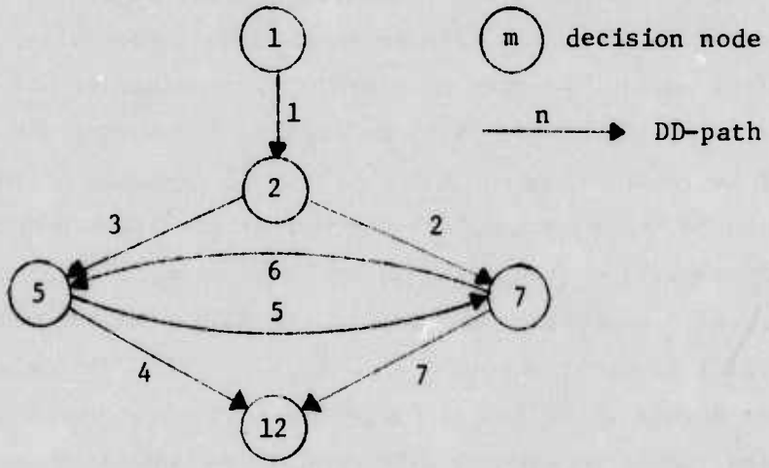
The problem which arises can best be illustrated by means of an example. Figure 4.1 shows the source code and (reduced) program graph for the FORTRAN FUNCTION subprogram INDEX, which returns the index of the first occurrence of the data item R in an array, A, having N entries. RXVP structural analysis of this module identified four level-0 paths, consisting of the DD-path sequences 1-2-7, 1-2-6-4, 1-3-4, and 1-3-5-7. The iteration represented by the DD-path sequence 6-5 was not identified as a level-1 path (and could not be according to the definition). In this instance, the failure of RXVP to reflect a possible iteration was easy to see. In more complex situations (e.g., if DD-paths 6 and 5 were more complicated subgraphs and/or the structure resulted from control constructs other than the DO statement), such an omission might not be as readily apparent.

Paige [7] points out that for programs in which only the three control constructs IF-THEN-ELSE, WHILE-DO, and sequential execution (shown to be sufficient by Bohm and Jacopini [11] for expressing any algorithm) are used, all level-i paths, i>0, are loops corresponding to WHILEs. There is, therefore, a direct correspondence between level-i paths and iteration levels in "structured" programs. While the level-i path concept can still be useful for (unstructured) FORTRAN programs, the relation between the structural notation and iteration in the program is much less direct.

Every cycle in the structural graph of a module represents at least a potential for iterative execution. If the aim is to permit decomposition of the program graph to reflect successive "levels" of iteration, it appears that either a better means of handling those DD-paths located at the intersection of cycles on different "levels" is required, or that logical considerations must be included in defining level-i paths. It is interesting to note that in the example given the level-0 path 1-2-6-4 is logically impossible. If notice is taken of the fact that executing DD-path 5 is prerequisite for executing DD-path 6, the latter DD-path can

Node	
[1]	FUNCTION INDEX (R,A,N)
	DIMENSION A(1)
[2,3]	IF(N.EQ.0) GO TO 20
[4]	DO 20 I=1,N
[5,6]	IF(A(I).EQ.R) GO TO 30
[7]	20 CONTINUE
[8]	I=0
[9]	30 CONTINUE
[10]	INDEX=I
[11]	RETURN
[12]	END

(a) Program Text and Node Assignments



(b) Reduced Program Graph

Figure 4.1 Array Search Function Example

only represent a repetition of the decision at node 5 and hence must lie on a level-1 path.

4.2.2 STEP 2 Discrepancies Noted

No discrepancies in STEP 2 operation were found during the project.

4.2.3 STEP 2 Statistics

RXVP STEP 2 required approximately 70000_{10} words of CYBER-74 main memory for execution. This includes provision of the Standard Print Commands for printing the library tables produced in STEPs 1 and 2.

An approximate upper bound on the number of words, S, of mass storage required for the complete RXVP random-access library file for most FORTRAN programs is given by:

S = 2000 + 3000(number of modules) + 60(number of statements).

4.3 RXVP STEP 3 (TESTING INSTRUMENTATION)

STEP 3 produces the instrumented version of the FORTRAN modules selected to be tested in subsequent STEP 4 execution. Control options enable the user to specify (1) whether or not the instrumented version of the module is to be listed, (2) whether or not a compilable deck is to be produced, (3) whether all DD-paths or just module ENTRYs are to be instrumented, (4) whether or not the standard set of instrumentation statistics is to be printed, and (5) the name of the trace data generation module, if the RXVP standard module (named TSPG871) is not to be used in STEP 4. STEP 3 manipulates a given input module to produce a logically equivalent version with software probes (CALLs to a trace data generation module) inserted in each DD-path (if that is the option selected). This instrumented version of the input module, written in compilable form on the file LPUNCH, is the normal principal output of STEP 3.

Users were generally satisfied with the features of STEP 3 (except for four important discrepancies to be discussed) and found it easy to use. One additional capability was suggested to improve the convenience of subsequent STEP 4 testing. Because of the way in which

STEP 4 controls the invocation of programs under test and recaptures control following their execution, any PROGRAM must be changed to a SUBROUTINE before being processed by RXVP if it is to be executed under STEP 4 control. The STEP 3 instrumentor could eliminate this requirement by automatically converting a PROGRAM card to a SUBROUTINE on the instrumented output file LPUNCH. In STEP 4 the user must supply a trivial driver routine (TSTPRG) to invoke the instrumented program via a SUBROUTINE CALL. This requirement could also easily be eliminated if STEP 3 provided at least the option to automatically generate the driver routine on the file LPUNCH, with the name of the program to be invoked supplied either in the option control statement or by recognition of the PROGRAM card.

4.3.1 STEP 3 Deficiencies Noted

No deficiencies were identified in using STEP 3.

4.3.2 STEP 3 Discrepancies Noted

(a) The instrumented modules produced by STEP 3 could not be compiled using the FTN compiler if the original module contained a DATA statement. FTN requires any DATA statement to follow all specification statements. LOGICAL variable declarations generated by STEP 3 in the instrumentation process are inserted preceding the first executable statement in the module. If a DATA statement is present in the original module, this causes a compilation error.

(b) Multiple ENTRY points are instrumented incorrectly. For a module having n ENTRYs, RXVP STEP 2 assigns DD-paths 1 through n to the ENTRY points in the inverse order in which they occur in the module. That is, DD-path n represents the primary module entry and DD-path 1 represents the last alternate ENTRY encountered in reading the module source code. STEP 3 instrumentation, however, causes the primary module entry point to report as DD-path 1 and the alternate ENTRYs to report with DD-path numbers reflecting the order in which the ENTRYs occur in reading the source code (i.e., STEP 3 numbers module ENTRY points inversely to STEP 2). As a result, execution coverage reports do not correctly reflect the portions of the module actually executed.

(c) A second discrepancy in instrumenting multiple-entry modules is not related to the numbering of DD-paths. Code representing an alternate ENTRY, k, in a multiple-entry module is often executed as a result of an

invocation through another ENTRY in the module. To prevent falsely reporting an invocation through ENTRY k in this situation, STEP 3 inserts logic intended to temporarily disable the probe in the DD-path assigned to ENTRY k. This probe, however, is not reenabled prior to exiting the module. As a result, once the module has been invoked via any entry point, subsequent invocations through any ENTRY other than the primary entry point will not report execution of the DD-path assigned to the ENTRY.

(d) RXVP uses the " character as a means of marking significant blanks within FORMAT statements in modules stored on the random-access library file. When the instrumented version of these modules is generated, STEP 3 restores the blank in place of the double-quote. This substitution is accomplished a little too zealously, however, with blanks substituted for all double-quotes, not just the ones originated by RXVP. As a result, Hollerith strings which were originally delimited by the " character become unrecognizable and cause errors in compilation.

(e) The final STEP 3 discrepancy noted causes no problem with any portion of subsequent testing activities. It merely reflects an innocuous abnormal condition. If the last non-blank character in any RXVP-formatted source code line (i.e., having tokens isolated from one another by blanks) occurs after Column 65, a spurious blank CONTINUATION card is generated on the file LPUNCH. This holds true for COMMENTS as well, although the spurious card is not a blank COMMENT but becomes a (blank) continuation of the first non-COMMENT card preceding the COMMENT.

4.3.3 STEP 3 Statistics

The primary statistics of interest when instrumenting a program for testing are the effects on the program's core requirements and execution time. These effects obviously depend on the nature of the program. The core expansion factor for a program consisting of many short DD-paths is going to be greater than the expansion factor for a program with a few long DD-paths. The effect on execution time is going to be greater for a program with high rates of iteration and short DD-paths than for a program with long DD-paths and little iteration. Appendix A reports the effects of instrumentation on the programs used in the investigation of RXVP characteristics. Execution time expansion factors ranged from 4.75 to 24.59 with no

clearly discernible characteristic value. The core expansion factors for the programs considered were much more consistent, with the average expansion of executable core due to STEP 3 instrumentation being by a factor of 2.4. Executable core is that portion of main memory occupied by a program exclusive of global and local data storage (i.e., not including I/O buffers, arrays, variables, and constant storage).

STEP 3 required approximately 51500_{10} words of CYBER-74 memory for execution.

4.4 RXVP STEP 4 (QUICKLOOK ANALYSIS)

The purpose of STEP 4 is to control the execution of a set of instrumented modules, to record the trace data generated during the execution, and to provide reports outlining execution coverage achieved by each invocation of the program under test and cumulatively for all invocations. Unlike the other RXVP STEPs, STEP 4 does not require access to the RXVP library file to accomplish its function. This makes it possible to test programs executing on a machine other than the RXVP host. Depending on the particular testing situation and machine to be used, some "customizing" of STEP 4 may be required for most efficient use. Sections 3.2.1 and 3.2.2 discuss the custom installation of STEP 4 on a Datacraft 6024/4 computer at AFAL and the procedure for using this installation in program testing. Experience with this customized version of STEP 4 will be discussed shortly.

The standard version of STEP 4 controls the testing activity in response to commands supplied by the user in the RXVP command stream. The program under test operates as an independent entity under control of STEP 4, reading its own inputs and producing its own outputs as usual. The command options permit the user to specify the instrumented modules for which trace data is to be collected, those modules for which individual execution coverage reports are desired, and the types of reports to be produced. After module selection and report options are set, the user supplies a series of Test and Test-Case Identifier cards. The Test Identifier is used to classify a

series of test cases which follows as a set. It is printed as a heading on each page of the reports generated for all test cases in the set. Each Test-Case Identifier serves to define a corresponding set of input data on the input file of the program under test. STEP 4 invokes the program under test once for each Test-Case Identifier found in the RXVP command stream. The corresponding input data supplied must be sufficient to drive the program under test to a termination. The Test-Case Identifier is printed as a subheading for the reports generated as a result of the corresponding invocation. When the command END SET is encountered in the RXVP command stream, the STEP 4 standard commands processor regains control and a new set of option settings may be made for subsequent tests.

In addition to being recorded on an execution trace file, trace data is accumulated by STEP 4 in a series of COMMON blocks. These blocks also contain the control option settings and working storage. Because their size is predetermined, they set limits on certain aspects of the testing activity. These blocks and their default constraints are summarized in Table 4.1. In the standard version of STEP 4 on the CYBER 74, the user can override the default values by providing a BLOCK DATA program defining the desired block storage capacity. This procedure is outlined in the RXVP User and Reference Manuals.

STEP 4 was found to be generally easy to use and the reports well formatted, easy to interpret, and adequate for the intended function of reflecting execution coverage. This STEP was, in fact, considered to be one of the best developed in the RXVP system. The function provided by STEP 4 was found to be of significant value to the testing activity, particularly when conducting tests in the role of acceptance tester. The principal benefit was an accurate record of exactly what had been tested by the various test case data-sets. While DD-path execution coverage is not in itself a sufficient testing objective, or even an adequate measure of the value of a particular test case, it does reflect the degree to which the logic has been

TABLE 4.1

DEFAULT STORAGE AREAS

Block Name	Contents	Default Maximum
MODNMS	Module names	10 modules
NMDDPS	Number of DD-paths per module	10 modules
LSTMOD	Modules selected for single module reports	10 modules
MODSPC	Module numbers	10 modules
INDEXS	Module data storage pointers	10 modules
ONETST	Single test DD-path counters	200 total DD-paths in all modules
DDPTST	Module total DD-paths executed per test	10 tests (10 modules)
INVTST	Module invocations, single test	10 tests (10 modules)
INVCUM	Module invocations, cumulative	10 tests (10 modules)
NOTHIT	DD-paths not hit working storage	100 paths not hit
CUMTST	Cumulative test DD-path counters	200 total DD-paths in all modules
DDPCUM	Module total DD-paths executed cumulative tests	10 tests (10 modules)

investigated and permits evaluation of the relative efficiencies of various data-sets in exercising the program. In addition to reporting testing coverage, the STEP 4 reports in one case led directly to the discovery of a program error by disclosing that, although not evident from normal output, a program was not executing as presumed.

4.4.1 STEP 4 Deficiencies Noted

No deficiencies were identified in using STEP 4.

4.4.2 STEP 4 Discrepancies Noted

No discrepancies were discovered in STEP 4 operation.

4.4.3 STEP 4 Statistics

Using the default block storage capacities previously discussed, STEP 4 required approximately 16500_{10} words of CYBER-74 main memory (in addition to that occupied by the instrumented program) for control, data collection, and report generation routines and I/O buffers.

4.4.4 STEP 4' (Datacraft Testing Analysis)

While the remarks which follow apply directly only to the custom version of STEP 4 installed on the Datacraft 6024/4 computer at AFAL, they reflect considerations which are felt to be fairly common in testing programs on a relatively small computer system.

The principal requirement to be satisfied by the version of STEP 4 installed on the Datacraft was to minimize the memory overhead connected with execution of instrumented code. The programs to be tested required a significant percentage of the Datacraft memory for execution, even when uninstrumented. When fully instrumented, they could not be loaded. In view of this situation, it was necessary to test the programs by instrumenting a subset of the program modules, executing the program with a given test case data-set, recording the results in a testing HISTORY file, repeating the execution of the program with a different subset of modules instrumented, and combining the results with the data already on the HISTORY file. (For one of the programs tested, ten such

executions were required for each test case data-set to provide information equivalent to one fully instrumented execution.) Because of this mode of operation, the multiple invocation feature of STEP 4 was discarded as superfluous in STEP 4'. The STEP was partitioned into an execution/trace-data-generation step and a report-generation/HISTORY-file-update step. A modified version of the trace data generation module, TSPG87, and a trivial driver program were the only additional components loaded for execution with the instrumented code. The trace data was not accumulated in memory as in the standard STEP 4, but only recorded on the trace file. In this manner, the core overhead associated with execution of the instrumented code was reduced to less than 1000_{10} 24-bit words.

Report-generation/HISTORY-file update ran as a separate job using the trace and HISTORY files. The reports generated were the same as those provided by the standard STEP 4. Control options were also essentially the same, although the command format was different. In addition, options dealing with the use and update of the HISTORY file were provided. The COMMON blocks previously discussed for accumulating trace data and setting control options were part of the report-generation portion of STEP 4'. Because the programs under test (even when partially instrumented) exceeded the default constraints, block storage had to be increased. The Datacraft loader, however, requires that all modules referencing a COMMON block specify the same length for the block. The BLOCK DATA program method for overriding default storage values was therefore not directly applicable, although a rather devious means of using it was devised.

Another problem was encountered in recording the trace file. The standard trace data generation routine, TSPG871, writes a formatted, unblocked record on the trace file each time it is called by a software probe in the instrumented code. Using this method of recording trace data, a 2400 foot reel of magnetic tape could hold approximately 30000 trace records. The largest working

storage area available on the Datacraft disk could hold even less. For the programs under test, with only a portion of the modules instrumented, trace files consisting of up to 650000 records were not uncommon with certain test case data-sets. (This was true even when repeated records caused by iteration on a single DD-path were eliminated.) The trace data generation module was rewritten to block and buffer the data to the trace file. As a result, a reel of tape was capable of holding over 750000 trace records and total execution time was greatly reduced.

After some experience with the testing procedure of combining the results of several partially instrumented executions, the need for a HISTORY file editing and summary capability was recognized. A program providing the desired features was written, improving the convenience and flexibility of the testing process. In summary, STEP 4' testing activities on the Datacraft computer were more of a problem than corresponding activities on the RXVP host machine due primarily to two difficulties:

(1) a much less efficient testing procedure made necessary by core limitations not experienced on the larger machine, and

(2) an inability to record the trace file from test executions in the original manner on the mass storage devices available.

If the system is to be used in testing software executing on other than the RXVP host machine, some detailed knowledge of the internal operation of STEP 4 is probably necessary to permit tailoring the STEP to meet various testing situations.

4.5 RXVP STEP 5 (DETAILED TESTING ANALYZER)

The function of STEP 5 is to provide more detailed information regarding testing coverage by combining the data recorded on the trace file as a result of an instrumented test execution in STEP 4 with data in the RXVP library file reflecting the structure and complexity estimates of the program under test. Control options permit the user to define the set of STEP 5 reports to be produced and the modules to be included in the coverage analysis.

The new information provided by STEP 5 consists principally of describing execution coverage in terms of percentages of module statements and complexity exercised, and identifying the correspondence between the DD-paths executed and level-i path classes in the module. In addition, STEP 5 provides the capabilities to print the execution trace file, yielding a record of DD-paths traversed in execution order, and to present this same information in terms of a trace of the order in which level-i paths were completed by the flow of execution. Using the program graph presented in Figure 2.3 as an example, and numbering level-i path classes in the order in which they are listed in the upper-right portion of that Figure, an execution flow represented by the DD-path sequence 1-4-7-9-8-11 would be reported as a level-i path class trace in the order 5-4-1.

Because the level-i path (class) concept permits describing execution flows in terms of combinations rather than sequences of level-i paths (or classes), this latter feature was not considered to be particularly valuable as a trace of program execution, particularly in modules having complicated structures and several iteration levels. Considered as a report of the combination of level-i path classes exercised, the same information could be presented in a much more readable form.

Only two of the six available STEP 5 reports were considered significant. One was the DD-path trace previously mentioned. The other was a cross-reference table displaying the membership of each DD-path in the various level-i path classes, and whether or not that DD-path had been exercised. Of marginal interest was a summary report reflecting, among other things, the number and percentage of statements exercised. The other reports were considered to be either of little value to the testing activity or redundant in view of the STEP 4 reports.

The thing most users wanted to see, following an examination of the STEP 4 reports, was the set of code comprising the DD-paths not exercised. To get this information the ASSIST,PROPERTIES,

DDPATHS,.... command of STEP 10 was used. STEP 10, in fact, often became the de facto "detailed testing analyzer." This preference was a reflection of the desire to see testing results in terms of the program itself rather than in terms of an abstraction of the program (i.e., the DD-path structural model). The STEP 10 Properties Report was ideal for this purpose, with the exception of the composite predicate portion of the report which was somewhat inappropriate when parallel DD-paths were included in the set not exercised. Because of this experience, it is felt that the capability to produce some version of this report automatically, selecting DD-paths to be included based on the trace file data, would improve the utility of STEP 5 in the testing activity.

4.5.1 STEP 5 Deficiencies Noted

No deficiencies (other than that previously implied by the suggested improvement) were noted in using STEP 5.

4.5.2 STEP 5 Discrepancies Noted

(a) RXVP documentation and STEP 5 report headings indicate that the data presented as a result of the STEP 5 commands ANALYZER,LIP TRACE and ANALYZER,TABLE is in terms of level-i paths. In fact, the data is in terms of level-i path classes.

4.5.3 STEP 5 Statistics

STEP 5 required approximately 69500 words of CYBER-74 memory for execution (if the Standard Print Commands were included).

4.6 RXVP STEP 6 (SELF-METERING INSTRUMENTATION)

STEP 6 provides the capability to instrument selected modules to collect data reflecting the behavior or performance of each individual statement during execution of the module. The output of the STEP is the instrumented version of the selected module, which is written in compilable form on the file LPUNCH, together with a report giving the options in effect for the instrumentation. User options permit specifying the name of the file on which the instrumented code is to be written, the name of the file on which the execution data

generated by the instrumented code is to be recorded (default name LTEST), and whether or not statistics on the values of variables resulting from execution of ASSIGNMENT statements are to be recorded.

Unlike STEP 4 execution of programs instrumented at the DD-path level, no reports are generated in conjunction with the execution of programs instrumented at the statement level. Reports are generated instead by STEP 7 from the data file recorded during the instrumented execution. Only the data recording routines called by the statement-level software probes and (because STEP 4 requires PROGRAMS to be converted to SUBROUTINES prior to being processed by RXVP) a trivial driver routine need to be loaded with the instrumented code for execution.

If STEP 3 were capable of automatically converting PROGRAMS to SUBROUTINES during DD-path level instrumentation as suggested in Section 4.3, the requirement for a driver routine to execute statement-level instrumented code would be eliminated.

4.6.1 STEP 6 Deficiencies Noted

No deficiencies were identified in using STEP 6.

4.6.2 STEP 6 Discrepancies Noted

(a) The instrumentation of IF statements, as presently accomplished, may result in an instrumented version of the module which is not logically equivalent to the original module. Specifically, if the predicate of an IF statement contains a FUNCTION invocation, and if the FUNCTION has memory (i.e., if the value returned is a function of previous as well as the current invocation), then the instrumented code is not logically equivalent to the original module because the instrumentation produces extra invocations of the FUNCTION.

(b) Double-quote marks in FORMAT statements are replaced by blanks, just as discussed in Item (d) of Section 4.3.2, resulting in unrecognizable Hollerith strings that were originally delimited by the " character.

(c) As in Item (e) of Section 4.3.2, if the last non-blank character in any RXVP-formatted source statement (other than COMMENTS) occurs after Column 65, a spurious (blank) CONTINUATION card is generated. RXVP-formatted COMMENTS, however, are truncated to Column 71 with no continuation.

4.6.3 STEP 6 Statistics

STEP 6 required approximately 47000_{10} words of CYBER-74 memory for execution. Executable core expansion factors due to statement-level instrumentation ranged from 2.8 to 4.9 for the set of programs considered, with the average module expansion factor being about 4.4. Execution time expansion factors ranged from 8.1 to 27.9.

4.7 RXVP STEP 7 (SELF-METERING ANALYSIS)

STEP 7 produces the statement-level execution report describing the execution of modules previously instrumented by STEP 6. The trace file recorded during the instrumented execution and module information from the RXVP library file are used in generating the report. The report takes the form of an annotated listing of the module, with the following information supplied:

(1) Statement number

(2) Statement text

(3) Statement execution count

(4) Count (and percentage) of TRUE evaluations of logical IF-statement predicates, and the result of the final evaluation

(5) Count of alternate branches taken from arithmetic IF-statements

(6) Number of times the module was invoked, the number of executable statements it contains, and the number actually executed.

If the module was instrumented appropriately in STEP 6, the initial, final, minimum, maximum, and average values of variables resulting from execution of ASSIGNMENT statements may also be reported. A report may be generated for every invocation of the module, to summarize the results of all invocations, or both. User control options also permit specifying the modules for which the reports are to be generated.

STEP 7 was found to be easy to use and the reports, as intended, to provide more detailed computational information than was available from STEP 4 and 5 reports. Because of this fact, STEPs 6 and 7 are

far more appropriate for program development and debug testing than are STEPs 3, 4, and 5.

With regard to using STEPs 6 and 7 in these early program testing activities, there was one suggested additional capability within the present context of the STEPs: the ability to selectively extract trace file information to produce a detailed record of the activities of specified statements during execution.

4.7.1 STEP 7 Deficiencies Noted

No deficiencies were noted in using STEP 7.

4.7.2 STEP 7 Discrepancies Noted

No discrepancies were found in STEP 7 performance.

4.7.3 STEP 7 Statistics

STEP 7 required approximately 45000_{10} words of CYBER-74 memory for execution.

4.8 RXVP STEP 8 (STATIC ANALYSIS)

STEP 8 provides a collection of procedures for the static analysis of module code and produces a series of reports useful in program development, testing, and documentation activities. The reports are generated from the RXVP library file produced in STEP 1. (The structural information added by STEP 2 is not required.) STEP 8 services can be classified into three functional categories: (1) extended intra-module compiler checks/reports, (2) isolation of selected functional classes of data and statements, and (3) inter-module invocation checks/reports.

Included in the first category are: an enumeration of statement types used in the module, a variable cross-reference table indicating how and where the variables are used, a check of all array references for dimensional conformance with the array declaration, a check of all expressions for mode conflicts, a report describing the formal parameters of the module, and a list of all variables not explicitly typed in the module declarations. The second category consists of: identification of all local variables and constants, identification

of the module "communication space" (i.e., the variables appearing in the argument list and COMMON variables used), identification of all constants and variables used in predicates, identification of the "local memory" (defined to be those local variables used in predicates) of the module, identification of all product expressions, identification of all denominator expressions, a list of all READ statements with the associated FORMAT statements, and an outline of all DO-statement nests. The third category provides a check of all module invocations to determine that actual and formal parameters conform in number, type, and dimension. It also can provide, in either tabular or graphical format, the invocation tree rooted at a selected module. These checks/reports can be selected by the user either individually or in predefined groups.

The services provided by STEP 8 were useful in examining individual modules and particularly in determining the relations among several modules. With further refinement and enhancement, the variety of STEP 8 functions was felt to have even greater potential benefit.

One area of suggested improvement deals with determining module connectivity. The present module connectivity analysis is distributed over three STEP 8 functions: invocation tree generation, communication space definition, and invocation parameter conformance checks. For systems having large numbers of modules and extensive COMMON data, a report reflecting module connectivity as a function of data usage as well as control flow would be of considerable use in evaluating the effect of changes within a module on the rest of the system. The information necessary to produce this report is presently available in the RXVP library file, but a new STEP 8 function would be required to correlate it appropriately. A reverse invocation (called-by) tree would similarly be useful when changes in a module (e.g., number or function of parameters) make it necessary to identify all users of the module.

An additional suggested static check, often not difficult to accomplish, is for potential use of an undefined DO-loop index

following termination of the loop. This is a reasonably common oversight when program logic permits an early exit from the loop as well as loop termination.

Finally, the "local memory" of a module is defined to be the subset of local variables used in predicates. The command STATIC, MEMORY causes this subset to be reported. The rationale for the definition is that local variables used in predicates have the potential for retaining information from one invocation to the next which could affect the flow of computation. A local memory report can be quite valuable, particularly in development/debug testing activities, but the definition of local memory currently used is not felt to be sufficient for the intended purpose. Any local variable used in any manner before being initialized by a function on the module communication space and/or constant terms, is in fact a part of the local memory of the module.

4.8.1 STEP 8 Deficiencies Noted

(a) The formal parameter report produced as a result of the command STATIC,PARAMETERS indicates the use of each formal parameter as "PASS-IN". Formal parameters which appear only on the left side of ASSIGNMENT statements within the module are obviously "PASS-OUT" in nature. (The variable cross-reference report indicates that the information necessary to correctly reflect formal parameter usage is available.)

(b) In the report defining the module communication space, the number of formal parameters is reported, but not their names. COMMON variables used within the module are reported simply as "USED". It would be more useful to know how they are used (e.g., ASSIGNED, READ, or both). The cross-reference report again indicates that the information necessary to report this is available. (If the module communication space report were appropriately enhanced, the formal parameter report could be eliminated. It is felt that this would improve the system utility by reducing the effort required to obtain and review the desired information.)

(c) The name of a FUNCTION does not appear in the communication space report for that module, although it represents a means whereby the module communicates data externally. Likewise, FUNCTION references do not appear

in the communication space of invoking modules, although they represent a means by which the invoking module acquires external data. Variables involved in I/O operations are also excluded from the module communication space. Although FUNCTION names and I/O variables represent different classes of data communication than COMMON variables and formal parameters, including them in separate sections of the communication space report would provide a more complete picture of the module's connectivity with the external data environment.

(d) In the variable cross-reference report, array subscript variables are given the inappropriate usage tag "READ/PASS".

(e) Variables used in decision predicates are also tagged "READ/PASS" in the cross-reference report. A separate report (produced by the command STATIC,PREDICATES) correctly lists all variables used in predicates. The cross-reference report should reflect a consistent usage for these variables.

(f) There is, apparently, an undocumented limit on the number of variables which can be reported in the variable cross-reference report. A module containing 521 symbols caused RXVP to abort due to excessive data base accession errors when the cross-reference report was requested. No attempt was made to determine this limit, although it is known to be greater than 198 symbols.

4.8.2 STEP 8 Discrepancies Noted

(a) A DO-loop index variable previously used as the index to another loop is given the usage tag "READ/PASS" in the second DO statement rather than the correct tag, "DO-INDEX".

(b) When a FUNCTION uses the variable having the FUNCTION name as an argument in the parameter list for the invocation of another module (e.g., when passing the value the FUNCTION attained on the previous invocation), the invocation check report produced by the command STATIC,CALL CHECK reports the argument as having unknown form even though the mode of the FUNCTION is known.

(c) When an array element is passed as an argument in a parameter list, a parameter mode error is generated in the invocation check report, even if the mode of the array conforms with the mode of the corresponding formal parameter in the called module.

(d) Alternate ENTRY points are reported as unknown in the invocation check and invocation tree reports, even though identified as type ENTRY in the symbol table for the multiple-entry module.

(e) COMMON variables used in PROGRAM modules are not reported in the communication space report for the PROGRAM. (Recall that a PROGRAM need be changed to a SUBROUTINE only if it is to be executed under STEP 4 control.)

(f) Assignment of a value to a previously unused variable by an IF-ASSIGNMENT statement causes a "USE-BEFORE-SET" warning flag in the cross-reference report.

(g) A variable set by an ASSIGN statement is given the usage tag "READ/PASS" and, if previously unused, causes a "USE-BEFORE-SET" warning flag in the cross-reference report.

(h) In the enumeration section of the READ statement report, the number of conditional READ statements is included in the value reported as the number of regular READ statements.

(i) An integer variable used as a FILENAME or as the object of an ASSIGN statement (i.e., receiving the statement label), then subsequently used in arithmetic computations, is defined twice in the module symbol table and cross-reference report. Explicit typing information is ignored in reporting the arithmetic use of the variable. (The discrepancy here is probably in STEP 1. The results only manifest themselves as indicated in STEP 8.)

4.8.3 STEP 8 Statistics

STEP 8 required approximately 73000_{10} words of CYBER-74 memory for execution. This includes provision of the Standard Print Commands for printing the RXVP library tables.

4.9 RXVP STEP 9 (TESTING GUIDANCE)

STEP 9 provides a series of reports, based on the iteration structure of a module, intended to outline a strategy for achieving the testing goal of 100% DD-path execution coverage. The thesis on which the strategy is based is that DD-paths residing at the highest possible iteration levels are the most efficient targets of testing activities because "generating a testcase to reach as deeply as possible into the iteration structure will assure that as much collateral testing as possible is achieved" [12]. The principal

report depicting this strategy (produced by the STEP 9 command GUIDE,TESTGUIDE) consists of an expansion of the level-i path class tree for the module. This tree represents the ancestry relationships among the different level-i and level-i+1 path classes (i = 0,1,2,...). The level-i path class tree for the program structure of Figure 2.3 is shown in Figure 4.2. The execution resulting from any invocation of the module can be described by a traversal of the tree from the root to some terminal branch. The terminal branch represents the highest iteration level-i path class involved in the execution. The TERMINAL BRANCH TEST GUIDE report consists of a series of "Testcase Set" descriptions, one for each terminal branch level-i path class in the tree. Each description reflects the possible structural combinations of path classes of different levels which provide access to the specific terminal branch path class (i.e., the different possible paths through the tree from the root to the terminal branch path class). By selecting one of these possible combinations and supplying testcase data resulting in its execution, the user is able to access the DD-paths of the "deepest" iteration level-i path classes in the module (i.e., the terminal branches of the level-i path class tree). Assistance in constructing appropriate testcase data to force execution of the selected tree traversal is provided by STEP 10 reports.

Other STEP 9 reports provide additional information about the level-i path classes in a module, the properties of DD-paths, and the relationships between DD-paths and their successors. To remain within the core limits established locally for daytime operations, only the TESTGUIDE report was included in the version of STEP 9 installed on the CYBER-74. Most of the information available in the other STEP 9 reports was available in slightly different format in STEP 10 reports.

The utility of the guidance provided and how it should be applied were not obvious to most users of the system. STEP 9 was probably the least well received portion of RXVP, both in terms of

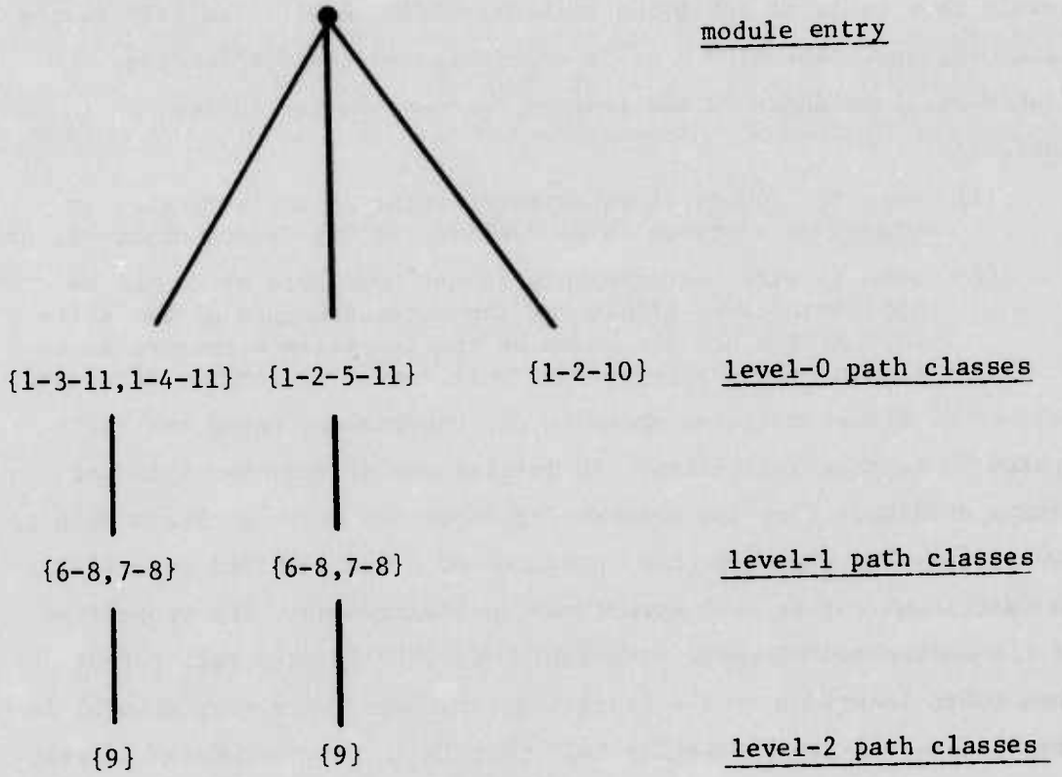


Figure 4.2 Level-i Path Class Tree Example

understanding the information presented in the output and in terms of testing strategy adopted (i.e., accessing the highest iteration levels as a means of achieving testcase efficiency). The RXVP User's Guide [9] describes STEP 9 as "a sophisticated tool" affording "substantial guidance in the testing process" in the following instances:

(1) when the number of untested DD-paths is so large that an organized approach to exhaustive testing is not apparent, or

(2) when initial testcase data is not available or cannot be applied because either the input requirements of the software system are not known or the iteration structure is so complex that appropriate inputs cannot be readily developed.

Neither of these instances arose in our experiences using the RXVP system in testing activities. An initial set of testcase data was always available from the program developer and we were always able to devise tests of the unexecuted portions of a module based principally on functional rather than structural considerations. The properties of the unexecuted DD-paths (available via STEP 10 reports), rather than their locations in the iteration structure, were most helpful in this regard. It was generally felt that if it were necessary to rely principally on structural information to devise tests of a module, it might be more difficult to assess the functional correctness of module activity and computational results in response to the testcase.

The principal benefit of iteration structure information in devising testcases derived, in our experience, from the correspondence often found between level-0 path classes and different functional flows through the module. Except for this aspect, we do not feel that the full utility claimed for the level-i path iteration structure model in developing testing strategy was demonstrated by our experience. This, of course, could be attributed to the particular testing activities conducted, the level of sophistication of the users of the tool, the model itself, or some combination of the three.

4.9.1 STEP 9 Deficiencies Noted

No specific deficiencies were identified in our use of STEP 9.

4.9.2 STEP 9 Discrepancies Noted

(a) In the section of the TESTGUIDE report which describes terminal branch level-0 path classes, the number reported as the "TITAL (sic) NUMBER OF DISTINCT LEVEL-0 PATH CLASSES" is actually the number of distinct level-0 paths in the set of terminal branch level-0 path classes.

4.9.3 STEP 9 Statistics

With only the TERMINAL BRANCH TESTGUIDE report option available, but including the universal Standard Print Commands, STEP 9 required approximately 58000 words of CYBER-74 memory for execution.

4.10 RXVP STEP 10 (TEST CASE ASSISTANCE)

Unlike the other RXVP STEPs, STEP 10 is not intended to provide general reports reflecting characteristics of entire modules or groups of modules. It is, rather, designed to assist in detailed examinations of selected portions of the DD-path structure of a module. Specifically, the information provided is intended to assist in the construction of test case data to access unexercised DD-paths within the module.

STEP 10 reports are produced in response to commands of the form: ASSIST,<option>,<construction-specification>. <option> specifies the form the report is to take. <construction-specification> defines the set or execution sequence of DD-paths to be included in the report.

The two most commonly used <option> specifications are PICTURE, which results in a stylized directed graph representing the DD-paths defined by <construction-specification>, and PROPERTIES, which produces a report consisting of four parts:

(1) a list of the set or sequence of DD-paths defined by <construction-specification>

(2) the composite predicate, expressed in terms of module variables, which must be satisfied for the defined DD-path sequence to be executed

(3) the execution order sequence of statements from the module source text which comprises the defined DD-path sequence, together with the decision outcomes which must result if the indicated execution sequence is to be realized

(4) A detailed analysis of variable usage along the defined DD-path sequence.

There are two categories of <construction-specification>. One category defines a set of DD-paths having some specific property (e.g., those DD-paths comprising the level-0 paths in the module). The other category defines a DD-path execution sequence which includes a specific DD-path or level-i path class. It is this category of <construction-specification> which is most often used to examine the various means of accessing untested portions of a module and to define the appropriate testcase data to force the desired execution.

In defining the set/sequence to be included in the report, the "leader" DD-path of a DD-path class is normally selected to represent the class. Additional STEP 10 commands permit the user to specify that all DD-paths in a DD-path class are to be reported, or that a member of the class other than the "leader" is to be selected.

The command ASSIST,PICTURE,ALL DDPATHS, producing a directed graph of the entire module, was found to be extremely useful in understanding the overall structure of the module and in planning testing strategies to access untested portions of the module.

The command ASSIST,PROPERTIES,DDPATHS,<n,m_1,m_2,...m_n>, as discussed in Section 4.5, was found to be an excellent method of examining the characteristics of the n untested DD-paths m_1,m_2,...m_n following a STEP 4 test execution.

Because the process of selecting an appropriate sequence of execution to reach some untested portion of a module consists largely of examining the various alternatives presented by STEP 10, we believe that the utility of this STEP would be enhanced by an installation permitting its use in an interactive manner. This, unfortunately, was not the case with our installation, but it is understood that such interactive installations do exist.

The principal user dissatisfaction with STEP 10 resulted from the manner in which execution sequences for reaching specified module segments are generated. These sequences are based only on structural considerations, with no regard given to logical reachability.

Including logical considerations would significantly enhance the utility of STEP 10. Depending on the extent to which it is carried, it also might represent a considerable improvement task in terms of effort. The seed of such an effort may be found in the work on backtracking formula reduction reported by Miller and Melton [13].

4.10.1 STEP 10 Deficiencies Noted

(a) The purpose of STEP 10 is to offer services "providing assistance in the generation of testcases" [9]. The fundamental resource for accomplishing this function is the PROPERTIES report option. While explicitly stating every predicate to be satisfied internal to a module is of interest in many situations, it is generally not as useful in generating testcases as information in terms of appropriate states of the module input space. That is, the present level of information intended to assist in testcase construction is too low for that purpose. More backtracking to attempt to define an appropriate set of input conditions is required. The developments referenced by Miller and Melton [13] are obviously designed to address this deficiency.

(b) When investigating the properties of different sets of DD-paths from several modules (e.g., the untested DD-paths following a test execution), it is sometimes not immediately obvious which module is being considered in an individual report. In this case, it would be helpful to place the module name in the heading of the ASSIST,PROPERTIES,<construction-specification> report.

4.10.2 STEP 10 Discrepancies Noted

(a) In listing the properties of DD-path sets/sequences (via the ASSIST,PROPERTIES,<construction-specification> command) for several modules (selected by MODULE = <modname>), if a composite predicate is truncated in any report due to excessive length, the composite predicates in all subsequent reports are truncated, regardless of their lengths.

4.10.3 STEP 10 Statistics

STEP 10 required approximately 62000_{10} words of CYBER-74 memory for execution.

4.11 RXVP SYSTEM LEVEL CONSIDERATIONS

This section discusses certain system-level aspects of RXVP and its use. The first topic to be considered is the familiarization and training required to permit beneficial use of the system. A potential user has three major questions to be answered:

(1) What services does RXVP provide?

(2) How do I acquire those services?

(3) How do I best use those services in testing activities?

There are three sources of answers to these questions: RXVP user documentation, formal instruction, and experience. User documentation consists of a User's Guide [9] and a Reference Manual [10]. The User's Guide describes the ten RXVP STEPs and generally how they are intended to be used in support of testing activities. It is not a comprehensive description of RXVP capabilities, in that only the principal options of each STEP are presented. The User's Guide adopts a "cookbook" approach in describing how to access the RXVP services discussed. The Reference Manual provides a catalog of RXVP capabilities and a more complete discussion of the RXVP command language and its use in invoking the desired functions. It is not, however, oriented toward a discussion of the application of RXVP in testing activities. The two documents are thus complementary, with the User's Guide introducing the overall scheme of RXVP use and the Reference Manual providing the mechanics. As is, perhaps, to be expected with the first release of any document, there are several typographical and compositional errors in the User's Guide and Reference Manual. Some of these are described in Section 4.11.2. Taken together, the User's Guide and Reference Manual were considered to do an adequate job of answering the first of the questions posed above.

The RXVP command language was judged to be easy to learn and convenient to use. The information contained in the Reference Manual provides sufficient tutelage to permit a user to apply the language in directing RXVP functions. The Reference Manual, when supplemented with installation-specific documentation regarding deck structure,

file access, etc., provides the necessary answers to the second question.

The amount of training required to answer the last question varies with the type of RXVP services being considered. Based on our experiences, we have classified the major RXVP functions into three categories, according to the type of user and minimum degree of training we estimate they imply in order to make effective use of the information they provide:

(1) Category I services consist of STEP 1, STEP 2 (considered as a library construction activity only), STEP 6, STEP 7, STEP 8, and the Standard Print Commands for printing the module, the module symbol table, and the entry point table. A casual user, self-trained by reading the User's Guide, Reference Manual, and installation-specific documentation, could probably make effective use of these resources.

(2) Category II services consist of all the above plus STEP 2 (including DD-path information), STEP 3, STEP 4 (on the host machine only), STEP 5 (DD-path reports only), and the Standard Print Command for producing the DD-path report. A casual user, after studying the user documentation and receiving about 8 to 12 hours of formal training in directed graph structural modelling and its use in DD-path testing, would probably be in a position to use Category II services effectively.

(3) Category III services include the above, as well as STEP 2 (complete with level-i path and complexity reports), STEP 5 (all reports), STEP 9, STEP 10, and the Standard Print Commands for producing the module Summary and History reports and level-i path (class) report. To make effective use of this information, a total of around 20 to 28 hours of initial formal training is probably required. This training would include complexity measure computation, iteration structure modelling, and their applications in program testing strategies. Our experience indicates that rather frequent use of RXVP is also required to remain proficient in the use of these concepts.

In each of these categories, efficiency in using the RXVP services will obviously improve with experience.

As reported in Section 4.4.4, we feel that application of RXVP in testing activities on other than the RXVP host machine probably requires additional familiarization/training in the internal operations of STEP 4. This is particularly true if the machine on which the

testing is to be accomplished has limited memory and/or mass storage resources. An RXVP specialist to assist in non-host testing activities, as well as to provide guidance in other aspects of RXVP use, is considered to be advisable at any installation where continued use of the system is anticipated. This specialist should be an experienced user of the system, familiar with the methods by which the RXVP instrumentation generates execution trace data and the methods by which that information is reported.

When planning an installation of RXVP (or any other software system for that matter), it must be kept in mind that, in addition to physical resource constraints, local operating procedures and administrative policies can have a significant impact on the availability of the system to the user and, therefore, on its utility. RXVP, in comparison with other local data processing tasks, was found to be a voracious user of on-line random access storage. (An estimated bound on the amount of storage required for the RXVP random access library file is given in Section 4.2.3.) Because RXVP executions frequently resulted in library files which would cause the amount of on-line storage allocated for AFAL permanent files to be exceeded, it was often not possible to catalog a library file for future reference. Users instead often had to create a new library file for each separate RXVP processing run by executing STEPs 1 and 2. Aside from the nuisance involved, this was obviously expensive in computer time. As a result, users began to plan longer, more involved runs, trying to accomplish a maximum number of tasks in a single RXVP execution. The installation of RXVP as a series of overlays permitted this to be done very conveniently. The longer execution time required for each job, however, resulted in increased turn-around time because of the job scheduling algorithm used. (The combination of field length and execution time required for these longer runs frequently resulted in a single turn-around under day-shift scheduling criteria.) This practical lack of system availability reduced the utility of RXVP since information could often not be obtained at the rate it was

needed in support of testing activities. The impact of having to create a new library for each separate RXVP run was particularly detrimental in the area of test case assistance (STEP 10), which, as previously discussed, is essentially interactive in nature. In a more permanent installation, some accommodation could undoubtedly be reached which would alleviate the situation just described. We mention it here simply to illustrate the importance of considering the local operating environment when assessing the potential utility of RXVP.

The following two sections identify deficiencies and discrepancies found in RXVP functions at the system level and in the user documentation.

4.11.1 System-Level Deficiencies

(a) When several reports are generated for each of several modules during a single RXVP execution, locating a specific report for a specific module in the resultant output is often not convenient. (RXVP output tends to be voluminous.) Since RXVP is intended for use in examining large software systems, this is a disadvantage which should be corrected. A Table of Contents at the beginning of the output generated by each execution would be a most useful addition. Considering the manner in which the output print file is produced, generating such a Table should not be overly difficult.

(b) The System Wrap-Up Summary, produced at the end of each RXVP execution, reports certain statistics describing the modules contained in the RXVP library file (e.g., number of statements, number of symbols, number of DD-paths, etc.). A total value over all modules in the library is generated for some of these individual statistics. The total number of DD-paths in all modules of a program (which, generally speaking, constitutes a library file) would make it easier to determine when the STEP 4 default storage capacities need to be overridden in testing the instrumented program.

4.11.2 System-Level Discrepancies

(a) The heading for the report generated by the Standard Print Command PRINT,LIP describes the contents of the report as the LEVEL-I PATHS FOR MODULE <modname>. What is, in fact, reported is a collection of representative level-i paths (composed of "leader"

DD-paths), one from each level-i path class in the module. The RXVP Reference Manual describes this report as "...a detailed listing of the RXVP information describing each Level-i Path Class identified for the current module." This is also misleading, as the entire classes are not described. The report should reflect all DD-paths in each DD-path class (e.g., in parentheses following the "leader" DD-path of the class) contained in the level-i path classes of the module.

As a related general observation, a great deal of user confusion and uncertainty could be avoided by more careful use of the terms "level-i path" and "level-i path class" in both RXVP reports and user documentation.

(b) During iteration of a set of RXVP commands for a group of modules specified by the module selection command FOR MODULES=<mod1>,<mod2>,..., when continuation cards are required to complete the module specification, the system attempts to interpret the continuation cards as RXVP commands as each new module is selected and reports them as unrecognizable. Following this, the intended set of commands is processed correctly for each module in the group.

(c) The STEP 8 denominator and product expression reports are not documented in the Reference Manual. The MODE CHECK report is illustrated but there is no written explanation of the illustrations. (It appears, in this latter case, that a page of text was omitted prior to printing the manual.)

(d) In Table 4.1-2 of the Reference Manual, illustrating the method for overriding the STEP 4 default storage values, the COMMON block ONETST, containing the single test DD-path counters, is omitted. Likewise, the COMMON blocks CUMTST and DDPCUM are omitted from the list of default storage areas in Table 4.1-1.

(e) The required execution verb, STRUCTURAL, is missing from the command sequence in the second example of Section 2.3 of the Reference Manual.

(f) The method given in Section 4.2.1 of the Reference Manual for continuing the list of MODULES REPORTING in STEP 4 is incorrect.

4.11.3 System-Level Statistics

The approximate amount of memory required for each RXVP STEP, if installed as an independent entity, was previously given individually. In some cases, the Standard Print Commands (requiring

approximately 11500_{10} words) were included with the STEP. Installed as a series of overlays, a field length of less than 60000_{10} words was sufficient to permit access to all RXVP functions (except as discussed in Section 4.9).

The files containing the RXVP binary modules themselves occupied approximately 192000 words of mass storage.

5.0 SUMMARY AND RECOMMENDATIONS

In addition to providing information about the RXVP system specifically, the work accomplished during this project reenforced some previous perceptions of the testing process and the use of automated aids in that process.

First and foremost, the testing of computer software remains largely an art. Testing philosophies, resultant testing strategies, and degrees of confidence in the final product vary widely. If testing is to remain the principal means of establishing the quality of software systems (and all indications are that, at least for large systems, this will be the case for the foreseeable future), the real need is for the development of a general testing methodology, well grounded in theory and demonstrated in practice, which permits relating, in a quantitative manner, some measure of program "testedness" with some measure of the "reliability" to be expected of the product. This is no small order, but there are research efforts in this direction (notably [14] and [15]). Until this panacea appears, the utility of automated tools developed to aid in the testing process will depend to a great extent on congruence between the views of testing held by the tool developer and the tool user.

Regardless of the testing approach adopted, there are three obvious qualities necessary for any automated verification aid: (1) the tool itself must be well qualified, so that the user can have confidence in the completeness and the correctness of the information it provides; (2) the tool must be readily available to the user in the testing environment; and, (3) the tool must be convenient to use and provide easily interpretable results. When these three requirements are satisfied, and when the tool provides services consistent with the testing approach, an automated verification aid can be of significant benefit to testing activities, both in terms of efficiency and accuracy.

In our experience with RXVP in software testing, we noted three classes of automated assistance which we feel could be of benefit in almost any testing environment (although not necessarily specifically

as implemented in RXVP, Level 1): (1) testing coverage services, reporting what portions of a program have been exercised and in what combinations, both for an individual test case and cumulatively for a set of test cases; (2) test case data generation services, analyzing the properties of program segments of interest and backtracking to indicate appropriate states of the program input space to access those segments; and (3) static analyses and reports reflecting both intra- and inter-module control and data flow, checking conformance with programming standards, reporting potential errors, etc.

Regarding RXVP, Level 1, specifically:

(1) The system provides an impressive array of services. It is new to operational use, however, and some of these services are better developed than others. There is a need for further refinement/improvement before the system achieves its full potential. The design is such that incorporation of these and future enhancements should be possible without major revision of the total system.

(2) A more elaborate syntax analyzer is required in STEP 1 to make the system compatible with specified FORTRAN dialects.

(3) The DD-path structural model is not difficult to fathom and provides a convenient basis for examining the execution coverage of program logic.

(4) The utility of the iteration structure model in testing strategy guidance and test case construction assistance has not been adequately demonstrated. As currently defined, the model can fail to reflect actual iteration in a FORTRAN module. It is, in addition, not for the casual use of the programmer since it requires some effort to learn and some continued application to retain proficiency in its use.

(5) The static analysis features of STEP 8 represent a set of potentially very useful services. There is, however, room for improvement in the implementation of several current functions. A few additional capabilities would, in our estimation, improve the utility of the STEP.

(6) There are critical errors in both the STEP 3 and STEP 6 instrumentors which require corrections.

(7) The core expansion produced by the instrumentation process can be a serious problem, particularly in testing programs close to the memory limit of the machine. It is often necessary to instrument and test a portion of a program (or even a program module) at a time, later correlating the results of the separate executions manually. In the same

vein, recording the execution trace file may become a problem when testing modules involving a considerable amount of iteration, particularly on systems having limited file space or which limit the number of file references.

(8) The execution coverage reports and statement-level execution reports are excellent. These are easily the best developed aspects of the system, although the STEP 5 (ANALYZER) reports offer little in addition to the STEP 4 (QUICKLOOK) reports as presently constituted.

(9) The STEP 9 testing strategy guidance was not particularly well received. It required too much knowledge of RXVP structural terminology and too much flipping of structural report pages to be conveniently used, and was based on a testing approach that was not widely accepted.

(10) The STEP 10 test case construction assistance represents a good start, but requires more development. In particular, logical as well as structural reachability needs to be considered and more backtracking capability provided if the assistance is to be of maximum benefit to the user.

(11) The RXVP library files required what, at this installation, is a considerable amount of on-line storage. The frequent inability to catalog library files for future reference resulted in a mode of operation which reduced the utility of the system.

(12) An RXVP specialist to assist users in applying the system is recommended for any installation where a significant amount of use is anticipated, particularly if testing on other than the RXVP host machine is involved.

Throughout this report it has been our objective to evaluate RXVP as it came to exist during the course of the project. The items mentioned herein reflect the status of our installation as of October, 1975. Any erroneous conclusions or inferences are a result of our still imperfect knowledge of the system, viewed from the perspective of a user. We hope the information presented will be of interest to both potential users and developers of this and other automated verification aids.

REFERENCES

1. Holland, J. G.
"Acceptance Testing for Application Programs"
Program Test Methods, W. C. Hetzel (ed.)
Prentice-Hall, 1973.

2. Gruenberger, F.
"Program Testing and Validating"
Datamation, 14:7, July 1968, pp 39-47.

3. Boehm, B. W., et.al.
Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980's (CCIP-85), Vol I,
AD742292, April 1972.

4. Reifer, D. J.
"Automated Aids for Reliable Software"
Proceedings of 1975 International Conference on Reliable Software
IEEE Cat. No. 75CH0940-7CSR, April 1975, pp 131-142.

5. Kennedy, J. E.
A Survey of Automated Computer Program Verification Tools,
Aerospace Corporation Report TOR-0075 (5112)-1, August 1974.

6. Ramamoorthy, C. V. and S. F. Ho
"Testing Large Software with Automated Evaluation Systems"
Proceedings of 1975 International Conference on Reliable Software
op. cit., pp 382-394.

7. Paige, M. R.
"Program Graphs, an Algebra, and Their Implication for Programming"
IEEE Transactions on Software Engineering, SE-1:3, September 1975,
pp 286-291.

8. Miller, E. F. Jr.
Methodology for Comprehensive Software Testing
ADA013111, February 1975.

9. RXVP FORTRAN Automatic Verification System, Level 1, User's Guide,
General Research Corporation, May 1975.

10. RXVP FORTRAN Automatic Verification System, Level 1, Reference Manual, General Research Corporation, May 1975.

11. Bohm, C. and G. Jacopini
"Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules"
Communications of the ACM, 9:5, May 1966, pp 366-371.

12. Miller, E. F. Jr., et.al.
"Structurally Based Automatic Program Testing"
prepared for EASCON'74, Washington, D.C., 7-9 October 1974

13. Miller, E. F. Jr. and R. A. Melton
"Automated Generation of Testcase Datasets"
Proceedings of 1975 International Conference on Reliable Software,
op.cit., pp 51-58.

14. Goodenough, J. B. and S. L. Gerhart
"Toward a Theory of Test Data Selection"
IEEE Transactions on Software Engineering, SE-1:2, June 1975,
pp 156-173

15. Brown, J. R. and M. Lipow
"Testing for Software Reliability"
Proceedings of 1975 International Conference on Reliable Software,
op.cit., pp 518-527.

APPENDIX A

RXVP Processing Statistics

Table A.1 presents statistics collected during RXVP processing of a set of FORTRAN programs of various types. The following RXVP control options were used in the processing reflected by the statistics:

STEP 1:	BASIC,CARD IMAGES=OFF. BASIC,COMMENTS=OFF.	
STEP 2:	STRUCTURAL,COMPUTE=FULL. STRUCTURAL,PRINT=FULL.	(i.e., Determine DD-paths, level-i paths, and complexity measures, and generate all STRUCTURAL reports.)
STEP 8:	STATIC,ALL.	(i.e., Generate all static analyses and reports.)
STEP 3:	INSTRUMENT,LIST=OFF.	
STEP 4:	QUICKLOOK,ALL MODULES. QUICKLOOK,ON.	(i.e., Generate all four QUICKLOOK reports for all modules reporting during the test execution.)
STEP 5:	ANALYZER,ALL MODULES. ANALYZER,ALL.	(i.e., Produce all available STEP 5 reports covering all modules in the program.)
STEP 7:	SMANALYZE,ALL MODULES. SMANALYZE,TYPE=SUMMARY.	(i.e., Produce a single, full statement-level report for each module summarizing the cumulative execution resulting from all invocations of the module.)

TABLE A.1

Sample Processing Statistics

Module (Program) Characteristics							RXVP Processing Time (seconds)								RXVP Library Size (words)	Instrumentation Expansion Factor			
																Core* Expansion Factor		Execution Time Expansion Factor	
Ident	# Stmnts	# DDPaths	# LIP Classes	# LIPs	Local Storage[a]	Core*[b]	STEP1	STEP2	STEP8	STEP3	STEP6	STEP4[c]	STEP5	STEP7		STEP5	STEP6	STEP5[d]	STEP6
A.1	8	3	1	2	3	13	.25	.31	.53	.34	.25					3.31	6.31		
A.2	8	1	1	1	3	25	.21	.30	.84	.28	.22					1.40	3.16		
A.3	88	58	7	53747720	28	373	2.79	1.60	2.55	4.15	1.92					2.46	2.67		
(A)	104	62			34	411	3.25	2.21	3.92	4.77	2.39	.72	1.81	2.13	15020	2.42	2.82	5.75	Note e
B.1	180	85	77	1710	436	458	7.88	19.97	12.14	6.23	3.79					2.47	4.58		
B.2	40	9	4	5	16	138	1.20	.68	2.31	1.15	1.06					1.79	4.46		
B.3	38	19	14	44	11	160	1.36	1.03	2.97	1.43	1.07					2.26	4.08		
B.4	62	27	9	308	19	265	2.27	1.35	5.22	2.43	1.91					2.20	4.10		
B.5	82	49	24	304	16	269	2.66	2.82	4.60	3.01	1.97					3.06	5.01		
B.6	135	57	28	336	20	624	4.99	4.30	11.28	4.45	3.58					2.03	3.90		
(B)	537	246			518	1914	20.36	30.15	38.52	18.70	13.38	2.13	18.75	9.79	42520	2.30	4.30	24.59	27.91
(C)	51	35	14	3536	21	165	1.5	1.2	1.72	2.44	1.1	4.19[g]	2.91[g]	6.21[g]	8020	2.86	3.38	Note f	Note f
D.1	101	23	12	36	71	432	4.34	1.33	5.98	3.10	2.46					1.38	2.41		
D.2	11	5	2	3	3	41	.25	.35	.89	.48	.32					2.44	3.41		
D.3	12	5	2	3	5	36	.36	.36	1.06	.48	.36					2.64	4.56		
D.4	76	17	8	9	72	256	2.74	1.18	4.52	2.06	1.83					1.67	3.76		
D.5	32	5	3	3	11	119	1.06	.61	1.54	.85	.94					1.43	4.27		
D.6	26	18	7	19	9	51	.77	.65	1.42	1.18	.68					4.24	6.41		
D.7	27	7	3	4	33	85	.88	.49	1.07	.85	.60					1.75	2.76		
D.8	6	1	1	1	3	11	.19	.30	.51	.24	.20					2.09	6.55		
(D)	291	81			207	1031	10.59	5.27	16.99	9.24	7.39	5.19	33.39	19.8	35520	1.73	3.35	9.39	10.44
E.1	93	8	2	9	123	341	3.76	1.19	7.63	2.22	2.09					1.18	3.22		
E.2	71	15	7	11	284	398	2.29	1.04	4.96	1.70	1.46					1.42	2.71		
E.3	23	5	3	3	14	119	.71	.46	1.82	.66	.63					1.35	2.92		
E.4	16	1	1	1	13	134	.54	.36	1.56	.45	.46					1.10	2.05		
(E)	203	29			434	992	7.30	3.05	15.97	5.03	4.64	1.18	4.43	7.8	20520	1.29	2.82	4.75	12.11
(F)	137	73	35	1666	1398	360	4.84	5.29	9.27	4.88	2.91	3.81	35.5	13.75	10520	2.52	4.18	7.38	8.09

TABLE A.1 (continued)

Sample Processing Statistics

Module (Program) Characteristics							RXVP Processing Time (seconds)								RXVP Library Size (words)	Instrumentation Expansion Factor			
																Core* Expansion Factor		Execution Time Expansion Factor	
Ident	# Stmnts	# DDPaths	# LIP Classes	# LIPs	Local Storage[a]	Core*[b]	STEP1	STEP2	STEP8	STEP3	STEP6	STEP4[c]	STEP5	STEP7		STEP3	STEP6	STEP3[c]	STEP6
G.1	36	12	5	7	79	102	.95	.76	1.66	1.03	.66					2.15	4.40		
G.2	100	77	32	36728	10	342	4.43	3.61	8.13	6.62	3.23					3.10	4.68		
G.3	8	1	1	1	1	19	.24	.31	.56	.25	.19					1.63	5.16		
G.4	8	1	1	1	1	19	.20	.30	.55	.33	.19					1.63	4.89		
G.5	24	15	6	14	9	58	.65	.93	1.15	1.17	.53					3.79	5.28		
G.6	22	9	5	5	4	78	.71	.53	1.30	.84	.51					2.13	3.35		
G.7	309	201	66	683	131	647	13.54	24.65	26.17	18.66	7.40					3.64	5.96		
G.8	169	91	474	212743	351	371	7.00	61.50	18.22	8.22	4.21					3.12	5.63		
G.9	154	55	26	23226	356	520	7.18	3.89	23.22	5.93	3.92					1.87	3.60		
G.10	25	9	3	6	118	49	.59	.49	.88	.71	.44					2.90	5.37		
G.11	7	1	1	1	4	29	.24	.35	.76	.25	.18					1.34	2.83		
G.12	20	11	6	6	9	34	.50	.47	.86	.69	.36					3.85	6.03		
G.13	12	7	3	3	7	23	.35	.39	.74	.52	.27					3.70	5.35		
G.14	10	5	2	3	4	25	.28	.36	.60	.40	.22					2.88	5.00		
G.15	20	5	2	3	8	50	1.11	.46	1.38	.83	.61					2.06	4.32		
G.16	39	17	2	129	27	127	2.04	.65	2.50	1.96	1.21					2.00	3.79		
G.17	15	3	1	2	13	11	.89	.34	1.64	.69	.49					3.55	8.55		
G.18	50	25	8	52	34	128	2.53	1.01	3.45	2.48	1.60					2.61	5.09		
G.19	40	23	7	146	24	74	1.94	.83	2.63	2.15	1.24					3.70	6.32		
G.20	108	51	18	2341	681	301	4.84	2.36	6.66	4.99	2.89					2.32	4.47		
G.21	6	1	1	1	0	30	.21	.26	.48	.24	.15					1.47	2.07		
G.22	45	33	6	4612	32	126	2.34	1.02	3.07	2.96	1.65					3.15	5.47		
G.23	76	41	27	939	30	171	3.25	2.39	9.37	3.53	1.96					3.08	4.84		
G.24	51	33	8	901	22	152	2.50	1.14	3.87	3.08	1.83					2.76	4.99		
(G)	1366	727			1955	3486	58.51	109.00	119.85	68.53	35.94	Note f	Note f	Note f	143020	2.80	4.88	Note f	Note f

Note a - Local Storage consists of all variable and constant data not accessible outside the module in which they are defined.
Note b - Core* denotes executable core, which is that portion of memory occupied by a module excluding local and global data, I/O buffers, parameter lists, etc.
Note c - Excluding execution of program under test; i.e., includes QUICKLOOK report generation and invocation control only.
Note d - Does not include report generation and invocation control.
Note e - Erroneous execution due to incorrect instrumentation by STEP 6 (see Section 4.6.2(a)).
Note f - Not measured.
Note g - Total for seven test cases.